University of Alberta

Non-deterministic Interaction Nets

by

Vladimir Alexiev

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Fall 1999

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-46797-X

Canada

University of Alberta

Library Release Form

Name of Author: Vladimir Alexiev

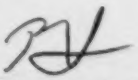
Title of Thesis: Non-deterministic Interaction Nets

Degree: Doctor of Philosophy

Year this Degree Granted: 1999

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.



Vladimir Alexiev
University of Alberta

Date: 29 June 1999

Procrastination is the thief of time.
—Edward Young

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Non-deterministic Interaction Nets** submitted by Vladimir Alexiev in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Jia-Huai You

Jim Hoover

Duane Szafron

Lorna Stewart

Curtis Hrischuk

James Harland

Date: July 2 1999

To my wife and my supervisor,
the two most patient people in the world.

Abstract

The *Interaction Nets* (IN) of Lafont are a graphical formalism used to model parallel computation. Their genesis can be traced back to the Proof Nets of Linear Logic. They enjoy several nice theoretical properties, amongst them pure locality of interaction, strong confluence, computational completeness, syntactically-definable deadlock-free fragments, combinatorial completeness (existence of a Universal IN). They also have nice "pragmatic" properties: they are simple and elegant, intuitive, can capture aspects of computation at widely varying levels of abstraction. Compared to term and graph rewriting systems, INs are much simpler (a subset of such systems that imposes several constraints on the rewriting process), but are still computationally complete (can capture the λ-calculus). INs are a refinement of graph rewriting which keeps only the essential features in the system.

Conventional INs are strongly confluent, and are therefore unsuitable for the modeling of non-deterministic systems such as process calculi and concurrent object-oriented programming. We study four different ways of "breaking" the confluence of INs by introducing various extensions:

IN with Multiple (reduction) Rules (INMR) Allow more than one reduction rule per redex.

IN with Multiple Principal Ports (INMPP) Allow more than one active port per node.

IN with MultiPorts (INMP) Allow more than one connection per port.

IN with Multiple Connections (INMC) Allow hyper-edges (in the graph-theoretical sense), *i.e.* connections between more than two ports.

We study in considerable detail the relative expressive power of these systems, both by representing various programming examples in them, and by constructing inter-representations that translate nets from one system to another.

We study formally a translation from the finite π-calculus to a system that we call *Multi-Interaction Nets*: MIN=INMP+INMPP. We prove the faithfulness of the translation to the π-calculus processes that it represents, both structural and operational (completeness and soundness of reduction). We show that unlike the π-calculus, our translation implements the *prefix* operation of the π-calculus in a distributed and purely local manner, and implements explicitly the distribution and duplication of values to the corresponding occurrences of a variable. We compare our translation to other graphical and combinatory representations of the π-calculus, such as the π-nets of Milner, the Interaction Diagrams of Parrow, and the Concurrent Combinators of Honda and Yoshida.

The original paper on IN (Lafont, 1990) states that INs were designed to be simple and practical; to be a "programming language that can be used for the design of interactive software". However, to date INs have been used only for theoretical investigations. This thesis is mostly devoted to a hands-on exploration of applications of IN to various "programming problems".

Acknowledgements

This work would have never been possible without the support and patience of my wife Dolia. Thank you for all that you do, and thank you for the wonderful child that you gave me. Our daughter Dianka was born on October 1, 1997, and to this day gives us immense joy, and some good fun as well.

I would like to thank my supervisor Dr Jia-Huai You for his guidance and patience during the long years of my graduate program.

I wish to thank Yves Lafont (whom I do not have the pleasure of knowing personally) for his invention of interaction nets, a formalism that is intuitive and pleasant to work with.

I gratefully acknowledge the financial support of the University of Alberta and the Isaac Walton Killam Memorial Trust throughout my graduate program.

The figures in this thesis were produced using a heavily-customized version of the dot graph layout program from AT&T. I wish to thank one of its authors Stephen North for his support and guidance.

Contents

1 Introduction 1

2 Interaction Nets 4
- 2.1 Definition of Interaction Nets . . . 4
  - 2.1.1 Example: Simple List Processing . . . 6
- 2.2 Linearity; Duplicator and Eraser Nodes . . . 8
  - 2.2.1 Example: Unary Arithmetics . . . 10
- 2.3 Useful Properties of Interaction Nets . . . 12
  - 2.3.1 Confluence and Determinism . . . 12
  - 2.3.2 Undesirable Situations: Blocking and Deadlock . . . 13
  - 2.3.3 Deadlock Avoidance . . . 14
  - 2.3.4 Structural Result: "Combing" a Deadlock-Free Net . . . 18
  - 2.3.5 Combinatorial Completeness of IN . . . 18
- 2.4 Applications of IN . . . 21
  - 2.4.1 SK Combinators . . . 21
  - 2.4.2 The λ-calculus . . . 23
- 2.5 Example: Infinite Streams in IN . . . 27
  - 2.5.1 Builtin Arithmetics . . . 27
  - 2.5.2 Infinite Numeric Streams . . . 28
  - 2.5.3 Stream of All Prime Numbers . . . 31
  - 2.5.4 Stream Merger . . . 33
  - 2.5.5 Stream Duplicator . . . 35
  - 2.5.6 The Hamming Sequence . . . 36

3 Non-Determinism in Interaction Nets 44
- 3.1 IN with Multiple (reduction) Rules (INMR) . . . 45
  - 3.1.1 Using Asymmetric Reflexive Rules to Capture INMR . . . 46
- 3.2 IN with Multiple Principal Ports (INMPP) . . . 48
  - 3.2.1 INMPP Example: Queue Merger . . . 48
  - 3.2.2 INMPP Example: Parallel Or . . . 49
- 3.3 IN with MultiPorts (INMP) . . . 50
  - 3.3.1 Uniform Treatment of Links . . . 50
  - 3.3.2 Arity Constraints . . . 51
  - 3.3.3 Rule Priorities . . . 53
  - 3.3.4 Example: Variable (Reference) . . . 54
- 3.4 IN with Multi-Connections (INMC) . . . 59
  - 3.4.1 Example: Concurrent Combinator Process Graphs . . . 60

4 Inter-representation of Models of Non-Determinism 62
- 4.1 Introduction . . . 62
  - 4.1.1 Completeness and Soundness . . . 63
  - 4.1.2 Example: Translating INMR to INMPP . . . 65
  - 4.1.3 Basic Ideas . . . 66

4.2 Separation Result: INMR is Weaker Than the Rest . . . 69
4.3 Representing INMPP as INMC: *Port Diamonds* . . . 72
4.3.1 The Grotesque Translation . . . 73
4.4 Representing INMP as INMC: *Port Diamonds* . . . 75
4.4.1 Basic Idea: Represent a Multiport as a Connection Point . . . 75
4.4.2 Port Diamonds . . . 76
4.4.3 INMP→INMC Interaction Steps (Protocol) . . . 78
4.5 Representing INMR as INMP: Self-Commitment Nodes . . . 80
4.6 Representing INMPP as INMP: Marker Nodes . . . 81
4.7 Representing INMC as INMP: Explicit Connector Nodes . . . 84
4.7.1 Avoid Premature Commitment Using Counting . . . 85
4.7.2 Merge Connected Connectors . . . 89
4.7.3 Complex Connector Arrangements . . . 91
4.7.4 Optimize the Translation Using Typing (cc-INMP Example) . . . 93
4.8 Future Work . . . 96

5 Representing the Finite π-calculus in Multi-Interaction Nets **97**
5.1 The π-calculus . . . 97
5.1.1 Definition of π-calculus Processes . . . 98
5.1.2 Structural Congruence . . . 98
5.1.3 Reduction Rules . . . 99
5.1.4 The Many Roles of Prefix . . . 99
5.2 Representing the π-calculus in MIN . . . 100
5.2.1 MIN_π Nodes . . . 101
5.2.2 MIN_π Labels . . . 102
5.2.3 Example: The Telephone-Email Communication . . . 102
5.2.4 Example: Blocking and Unblocking . . . 104
5.3 The Translation from π-calculus to MIN_π . . . 104
5.3.1 Complexity of the Translation . . . 107
5.3.2 Structural Correspondence of the Translation . . . 108
5.3.3 Inverse Translation . . . 109
5.3.4 Example: The Honda-Tokoro Construction . . . 110
5.3.5 Unnecessary Blocking in the π-calculus . . . 113
5.4 MIN_π Interaction Rules . . . 114
5.4.1 Send/Receive . . . 114
5.4.2 Input/Output (Link Migration) . . . 115
5.4.3 Blocking and Unblocking . . . 116
5.4.4 Example: Reduction of Prefix . . . 118
5.5 Operational Correspondence . . . 118
5.5.1 Soundness . . . 119
5.6 Discussion: Why Multiple Principal Ports? . . . 121
5.7 Related Work . . . 122
5.7.1 π-nets and Interaction Diagrams . . . 122
5.7.2 Concurrent Combinators . . . 122
5.8 Future Work: Replication and Choice . . . 124

6 Conclusion **125**
6.1 Directions for Future Work . . . 125
6.2 Contrubutions . . . 127

A Notations **128**

B Technical Note: How Was This Thesis Produced **130**

Bibliography **132**

List of Figures

2.1 General form of an Interaction Rule. 5
2.2 A Sample List, (1, 2, 3). 6
2.3 The List "Append" Operation. 7
2.4 Appending the Two Lists (1) and (2). 7
2.5 List "Reverse" Using an Accumulator. 8
2.6 Reversing the List (1, 2). 8
2.7 Nodes Duplicator δ and Eraser ϵ. 9
2.8 Possible Reflexive δ Interactions: $\delta \bowtie \delta \rightarrow |\,|$ and $\delta \bowtie \delta \rightarrow \bowtie$. 9
2.9 The Kind of the Reflexive δ Interaction Should Depend on the Reduction History. . 10
2.10 The Number 5 in Unary Notation. 10
2.11 Unary Arithmetics Represented in IN. 11
2.12 Even $2 \times 2 = 4$ is Not Trivial in IN. 12
2.13 A Simple Infinite Computation: flip-flop. 13
2.14 Creation of a Cyclical Wire. 14
2.15 Typical "Combed" Net. 18
2.16 The Interaction Combinators of Lafont. 19
2.17 The SK Combinatory System Represented in IN. 22
2.18 SK Reduction Represented in IN. 22
2.19 β-reduction in the Linear λ-calculus. 23
2.20 Translation of an Abstraction $\lambda x.b$. 24
2.21 The Translation of S and K Considered as λ-functions. 24
2.22 The Reduction of $[@(\Omega, \Omega)]$ (part 1). 25
2.23 Applying the Wrong Rule $\delta \bowtie \delta \rightarrow |\,|$. 25
2.24 The Reduction of $[@(\Omega, \Omega)]$ (part 2). 26
2.25 Built-in Arithmetic Operations. 27
2.26 Built-in Boolean Operations. 28
2.27 A Stream Responds to get Requests, Potentially *Ad Infinitum*. 28
2.28 A Stream Constructor, cons. 28
2.29 The Stream iota Returns a Sequence of Natural Numbers Starting From N. 29
2.30 Obtaining Two Numbers from iota(3). 29
2.31 The Stream Multiplier smul. 30
2.32 A Stream filter That Removes Multiples of N. 30
2.33 filt_2 Skips or Returns the Number h_1. 31
2.34 A Stream Returning All primes Using *Eratostene's Sieve*. 31
2.35 Obtaining the First Two Prime Numbers. 33
2.36 Stream Merger smrg (Step 1). 33
2.37 Stream Merger smrg (Step 2). 34
2.38 Stream Merger smrg (Step 3). 34
2.39 Stream Merger smrg (Step 4). 35
2.40 The Stream Duplicator sdup. 35
2.41 Duplicating the Stream/Sequence iota. 36
2.42 A Stream Solving the Hamming Problem. 37
2.43 Obtaining the First Four Elements From a Simplified Hamming Stream. 42

3.1 IN with Multiple (reduction) Rules (INMR). 45
3.2 Translating INMR to INAR: Basic Idea. 46
3.3 Translating INMR to INAR: Decision Making. 47
3.4 Translating INMR to INAR: Counting d Nodes Looking Towards d_i. 47
3.5 Translating INMR to INAR: Using the Decision d_i. 48
3.6 IN with Multiple Principal Ports (INMPP). 48
3.7 INMPP Example: Merger. 49
3.8 Parallel Or as an INMPP. 49
3.9 Parallel Or Connected to Both tt and ff. 50
3.10 IN with MultiPorts (INMP). 50
3.11 Edges are Migrated Between Ports of the Same Name. 51
3.12 Edges are Migrated Between Nodes of the Same Type. 51
3.13 INMP Arity Constraints. 52
3.14 Migration of the b.x Edge Bundle to a.x. 52
3.15 Migration of Two Edge Bundles Directed by Rule Priorities. 54
3.16 a Exhausts the Primary Edge Bundle, then Disappears. 54
3.17 A Variable var has State and Responds to get/set Requests. 55
3.18 Outcome Depends on Order of Interaction. 55
3.19 get/set Interact as Expected. 56
3.20 The Lower Reduction Leads to a Vicious Circle. 57
3.21 The Correct Way to Sequentialize Requests. 57
3.22 Duplicating a var Reference. 58
3.23 Example of Duplicating a var Reference. 58
3.24 Additional Rules for the acquaint Example. 58
3.25 A Sample INMC. 59
3.26 INMC Rules are Like Conventional IN Rules. 59
3.27 A Sample INMC Reduction. 59
3.28 cc Process Graphs. 60
3.29 Example of an INMC System: cc Process Graphs. 61

4.1 Translating INMR to INMPP Example. 65
4.2 INMR to INMPP Example: Auxiliary Rules. 66
4.3 INMR to INMPP Example: Main Rules. 66
4.4 INMR to INMPP Example: A Sample Reduction. 66
4.5 INMR Counter-Example in INMP, INMPP and INMC. 69
4.6 Counter-Example Rules 70
4.7 Hypothetical translation of the counter-examples in INMR. 70
4.8 INMR can implement both INMPP reductions. 71
4.9 Representing INMPP as INMC: the Basic Idea. 72
4.10 Translating INMPP to INMC. 73
4.11 Sample INMPP Rule to Translate to INMC. 74
4.12 Convert a Multiport to a Connection Point (Naive Approach). 75
4.13 The Naive Approach Doesn't Work. 75
4.14 Insert Unidirectional Nodes Which Point in the Same Direction as the Multiport. . . 76
4.15 The *Port Diamond* Configuration. 76
4.16 Translation of an INMP Node With Multiports. 77
4.17 Negotiation: Generation of Pre-commitment, Acknowledgment, Rejection. 78
4.18 Negotiation: Resolution of Individual Host Responses. 78
4.19 Rejection and Restoration of the Port Diamond. 79
4.20 Start Implementing the Interaction: Decrement the Principal Port Arities. 79
4.21 Passivate the Principal Edges of a; Produce RHS. 80
4.22 Representing INMR as INMP (Step 1) 80
4.23 Representing INMR as INMP (Step 2) 81
4.24 Representing INMR as INMP (Step 3) 81

4.25 Representing INMPP as INMP (Step 1). 81
4.26 Representing INMPP as INMP (Step 2). 82
4.27 Representing INMPP as INMP (Step 3). 82
4.28 Representing INMPP as INMP (Step 4). 83
4.29 Representing INMPP as INMP (Step 5). 83
4.30 Representing INMPP as INMP (Step 6). 83
4.31 Representing INMPP as INMP (Step 7). 84
4.32 Representing INMPP as INMP (Step 8). 84
4.33 Connector Node c. 85
4.34 Counting Connector. 86
4.35 Connector Commitment for Irreflexive Rules. 86
4.36 Decrementing Connector Counter i. 87
4.37 Decrementing Connector Counter j. 87
4.38 Connector Commitment for Reflexive Rules. 88
4.39 Decrementing Connector Counter. 88
4.40 Merging of Connection Points Through a Shortcut Edge. 89
4.41 Merging of a Newly-Introduced Connection Point with an Existing One. 89
4.42 Connector Commitment Produces a *Connector Source* cs. 90
4.43 Connector Merging: Migrate a_i Edge. 90
4.44 Connector Merging: Increase Count. 91
4.45 Migration of Connector Edges. 91
4.46 Representing cc Graphs as INMP (Step 1). 93
4.47 Representing cc Graphs as INMP (Step 2). 94
4.48 Representing cc Graphs as INMP (Step 3). 94
4.49 Sample cc-INMC Graph. 95
4.50 Corresponding cc-INMP net; Emulating the Upper cc-INMC Reduction. 95
4.51 Emulating the Lower cc-INMC Reduction. 96

5.1 The Basic Idea of the $\Pi \to$ MIN Representation: Translating the Reduction $c?x, c!v \to [v/x]$. 100
5.2 MIN_π Main Node Types. 101
5.3 MIN_π Auxiliary Node Types. 102
5.4 Translation of the Telephone-Email Process $t?x.x!m, t!e, e?y$. 103
5.5 Reduction of the Telephone-Email Process. 104
5.6 Simplified Connectivity Structure of $[a!e.c!d, c?x]$. 104
5.7 Adding a Blocker and Unblocker. 104
5.8 Translation of π-calculus Input Atom: $[a?x]$. 105
5.9 Translation of π-calculus Output Atom: $[a!x]$. 105
5.10 Translation of π-calculus Input Prefix: $[a?x.Q]$. 105
5.11 Translation of π-calculus Output Prefix: $[a!x.P]$. 106
5.12 Translation of the Honda-Tokoro Construction. 112
5.13 Minimized Variant of the Honda-Tokoro Construction. 114
5.14 (Hypothetical) Ternary Send/Receive Rule. 114
5.15 Send/Receive Rules. 115
5.16 Input/Output Rules. 115
5.17 Final Input/Output Rule. 116
5.18 Unblocking (step 1): Link Migration. 116
5.19 Unblocking (step 2): Removing Blockers. 117
5.20 Net (Aggregate) Result of $\mathsf{u} \bowtie \mathsf{p}$ Interaction. 117
5.21 The Reduction of $[a?p.Q, a!c.P]$. 118
5.22 Why Port o Needs to be Principal. 121
5.23 Why Port s Needs to be Principal. 122
5.24 β-Reduction as an Interaction Rule. 123
5.25 Interaction of Prefix $c?x.X$ with a Message in cc. 123

List of Tables

3.1 INMP Arity Constraints 52

4.1 Inter-representation of Non-Determinism: Basic Ideas. 69

5.1 MIN_π Port Names. 100
5.2 Translation of the Telephone-Email Process. 103
5.3 Proof of the Linear Complexity of Translation $\Pi \to \text{MIN}$. 108
5.4 Possible Edges in a Blocking Region 109
5.5 Reduction of the Honda-Tokoro Construction. 112

A.1 Notations 129

Chapter 1

Introduction

The *Interaction Nets* (IN) of Lafont (1990) are a graphical formalism for modeling parallel computation. Their genesis can be traced back to the Proof Nets of Linear Logic (LL). They enjoy various nice theoretical properties, amongst them pure locality of interaction, strong confluence, computational completeness, syntactically-definable deadlock-free fragments, combinatorial completeness (existence of a Universal IN). They also have nice "pragmatic" properties: they are simple and elegant, intuitive, can capture aspects of computation at widely varying levels of abstraction.

Conventional INs are strongly confluent, and are therefore unsuitable for the modeling of non-deterministic systems such as process calculi and concurrent object-oriented programming. We introduce four different ways of "breaking" the confluence of INs using two methods:

- Relaxing some of the constraints that conventional INs impose. We don't want to stray too far on that path, since we'd like our extended INs to be as close to conventional INs as possible.
- Allowing various extensions.

We study the expressive power of the extended systems in considerable detail, both by representing various programming examples in them, and by constructing inter-representations that translate nets from one system to another. We also study formally a translation from the finite π-calculus to one of our extended systems, which shows that concurrently interacting processes can be represented faithfully in non-deterministic IN.

Our interest in non-deterministic INs stems from the simplicity, elegance and nice theoretical properties of conventional INs, and from a desire to extend them so that they can also capture concurrent computation. Why introduce yet another formalism for concurrent computations, given the plethora of existing process calculi? Because unlike functional computation for which there exists wide consensus about the best ways to represent them, our understanding of the basic building blocks of concurrent computation is less advanced. Researchers are still trying to distill the essence of processes and concurrent interaction, and to determine what process constructors are necessary and should be "built-in". A further motivation comes from the tradition of LL, which has focused on the fine-grained aspects of computation, such as representing value passing and sharing explicitly. Most concurrency formalisms to date have left these aspects unaddressed, by using freely devices like substitution, which are global in nature (with respect to the term being substituted into). Most formalisms also use purely syntactic entities such as names essentially. We are able to capture one of the most well-known process calculi in our extended INs, by introducing a translation which addresses both of the above concerns.

Here is a road-map of the rest of this thesis:

Chapter 2 introduces conventional Interaction Nets. We first give a definition of IN, then proceed with various examples. Then we recount various important properties of IN, such as confluence, deadlock-freeness, combinatorial completeness. By showing a representation of SK

combinators and the λ-calculus in IN, we show their computational (functional) completeness. We finish the chapter with a substantial original example, the representation of potentially infinite data streams and their processing (Eratostene's Sieve and Hamming's Sequence).

Chapter 3 introduces four different ways of extending Interaction Nets with non-determinism:

IN with Multiple reduction Rules (INMR) allow more than one reduction rule per redex.

IN with Multiple Principal Ports (INMPP) allow more than one active port per node.

IN with MultiPorts (INMP) allow more than one connection per port.

IN with Multiple Connections (INMC) allow hyper-edges (in the graph-theoretical sense), *i.e.* connections between more than two ports.

In addition to definitions and numerous examples, we introduce several special topics regarding INMP (uniform treatment of multi-edges, arity constraints on the applicability of rules, rule priorities based on the most-specific applicable arity constraint), and we show that the Concurrent Combinators of Honda and Yoshida are an INMC system.

Chapter 4 studies the very natural question of what is the expressive power of the different extended systems that we have introduced, in relation to each other. We introduce inter-representations between these systems, and thus show which ones can capture which others, and at what price of complexity. It is not at all obvious which is the "right" way to introduce non-determinism in INs, and this comparative study gives us some insights into that question, and some confidence that three of the four ways are fairly reasonable.

First, we show that INMR is weaker than the other systems: it cannot capture them faithfully, under some reasonable definition of "faithful translation". Then we show how to represent INMP and INMPP in INMC. Then we show how to translate all the systems into INMP; this system seems to be the most convenient and generally-useful of the ones that we study, we have had most experience with this system. For all other translation directions, we provide ideas about how they can be implemented.

The translations are fairly involved, but we hope that the basic ideas are simple and clear. We don't give formal proofs of the faithfulness of our translations (that would require a framework of behavioral equivalence which is not yet developed for extended INs), but we hope that our exposition and examples are convincing enough.

Chapter 5 develops a translation of the finite monadic π-calculus into a system that we call Multi-Interaction Nets, MIN=INMP+INMPP. The π-calculus is arguably the most popular tool for the theoretical investigation of concurrent computations. We start by introducing the π-calculus, its structural congruence rules and its reduction relation. Then we motivate our translation by showing that the π-calculus *prefix* operation is global with respect to the prefixed process, and is "overloaded" with several different roles (communication, synchronization, value delivery). Our translation implements such prefix in a purely local and distributed way (distributed blocking), and separates the roles of synchronization and value passing. We also dispense with the operation of *substitution*, and thus show explicitly how a value reaches all the occurrences of a corresponding variable.

We first define the MIN node types, then the translation from π-calculus to MIN, and then the MIN interaction rules. We show both the static (structural) and the dynamic (operational) correspondence of our translation nets to the original processes that they correspond to. We study the size and time complexity of our translation, and show that they are linear. We give examples along the way, including the Honda-Tokoro construction which implements polyadic prefix in the monadic π-calculus. Finally, we compare our translation to related work, namely the π-nets of Milner (1994), the Interaction Diagrams of Parrow (1995), and the Concurrent Combinators of Honda and Yoshida (1994a).

Chapter 6 reiterates the contributions of this thesis, and gives some ideas about possible future work. This thesis is by its nature *exploratory*, it only "charts a map" of a new unexplored territory, that of non-deterministic Interaction Nets. The formal study of the properties of these nets, and especially how they relate to the pleasant properties of conventional INs, is left for the future.

The original paper Lafont (1990) on IN states that INs were designed to be simple and practical; to be a *programming language that can be used for the design of interactive software*. However, to date INs have been used only for theoretical investigations. This thesis is devoted mostly to a hands-on exploration of applications of INs (modified in various ways) to various "programming problems", including concurrency, communication, references (variables), and some aspects of concurrent object-oriented programming. Only chapter §5 on the representation of the finite π-calculus in Multi-Interaction Nets is really technical, the other chapters contain various constructions and examples, but few formal proofs.

Chapter 2

Interaction Nets

The *Interaction Nets* (IN) of Lafont (1990) are a novel model of asynchronous parallel computation which is simple, elegant, graphical and purely local. They represent explicitly all aspects of a computation, including the duplication/distribution of a value to a number of consumers. This makes INs suitable as an intermediate representation for other formalisms and allow us to measure the true complexity of a computation. Since the basic computation elements of IN are very simple, they are suitable for implementations in an abstract machine.

IN are designed to be useful both as an (abstract) programming language and as a useful intermediate representation. We find that INs are suitable to express naturally notions at vastly varying levels, starting as low as representation of integers, and finishing with higher-level concepts like communication and interaction.

INs were inspired by Linear Logic's Proof Nets (Girard, 1987), a representation of linear logic proofs that is free from any unnecessary sequentialisation. INs are a natural generalization of Multiplicative Proof Nets (MPN). Where MPN have only two fixed node types corresponding to the logical connectives *times* $\otimes$ and *par* $\parr$, INs allow a "user-defined" alphabet of node types. See (Lafont, 1995a) and (Danos and Regnier, 1989) for details.

This chapter presents an introduction to IN, and then surveys some theoretical results in a quite detailed way. There are several reasons why the chapter goes in such detail:

- To provide the reader with background material on IN, so that our thesis is as self-contained as possible.
- To give the reader a "feel for the spirit" of INs, and additional motivation why IN are interesting and important.
- To gather in one place results which have thus far been scattered in many papers, and in some cases to prove properties that have been known as "folk lore".

Nevertheless, only the following sections are required for understanding the rest of the thesis: §2.1, §2.2, §2.3.1, §2.3.2, §2.3.4. Section §2.4 shows the computation completeness of IN.

Most of the material in this chapter is not new. The substantial example on Infinite Streams §2.5 is a notable exception.

2.1 Definition of Interaction Nets

Interaction Nets (IN) consist of *nodes* (*agents*) connected by undirected *edges* (*connections*, *wires*). The points where edges enter nodes are called *ports*. In general, two ports are not interchangeable, so in order to be able to distinguish them, we denote them with different names or numbers.

Every port is connected to exactly one edge, and every edge is connected to one or two ports. Edges that are connected to only one port are called *free* or *dangling*. The unconnected (free) end

of such an edge is marked with a *variable*, which we denote with a capital letter. Variables are also called the *free ports* of a net. The set of free ports of a net is also sometimes called its *interface* or *boundary*, the intent being that it can be used to connect the net to another net.

Every node occurrence in a net belongs to a specific *node type*, which is characterized by a name and the multiset of the names of its ports (*signature*).

Definition 2.1.0.1 (Principality) *Every node has exactly one* principal *(*active*) port.*

A node may also have arbitrary many *auxiliary* (*passive*) ports. The arity of a node is defined as the number of its auxiliary ports.

Nodes can interact only through their principal ports. Two nodes connected by their principal ports form a *cut*[1] (*redex, live pair*).

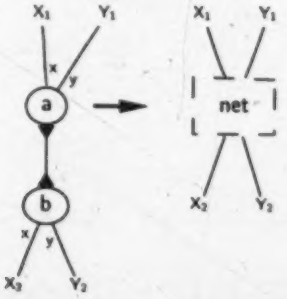
We denote node types with names in a sans-serif font (*e.g.* a). We denote a port p on a node a like this: a.p. We denote edges with a minus sign, *e.g.* an edge connecting ports a.p and b.q is denoted a.p−b.q. Sometimes, when it's clear from the context which ports we have in mind, we denote an edge simply with the node names, like this: a−b. We denote a cut edge with the join symbol: a ⋈ b. Most often we don't mention the ports of a cut edge, since they are always the principal ports of the two nodes in question.

A net with free ports is called *open*, and a net with no free ports is called *closed*. Usually only the evolution of closed nets is studied (that is why they are sometimes called *proper* nets), but open nets are important for the definition of interaction rules and rewriting (see below).

An IN system has a set of *interaction rules*, which are pairs of the form $L \to R$.

Definition 2.1.0.2 *The* left-hand side *(LHS) L of a rule always consists of two nodes forming a cut (redex)* a ⋈ b. *The other (auxiliary) edges of the two nodes are always free; they form the interface of the redex. The* right-hand side *(RHS) R is any (open) IN (including a net without any nodes), having the* same interface *as L.*

We denote interaction rules like this: [⋈ 2.1]



⋈ 2.1: General form of an Interaction Rule.

Nodes are indicated with ovals. Principal ports are indicated with bold triangles. Auxiliary ports are not indicated specifically, they are simply the points of entry of the edges into the nodes (and we often name them in order to be able to distinguish them). Whole subnets are indicated with large dashed boxes. For more details on the notations that we use, see §A.

A net evolves through a process of *reduction* (*rewriting, computation*) determined by the interaction rules.

Definition 2.1.0.3 (Rule Application) *A rule is applied to a net as follows:*

1. *An instance of the LHS of the rule (a redex) is located in the net.*

[1] This term comes from Linear Logic

2. *The redex is removed ("cut out") from the net.*

3. *The RHS of the rule is inserted ("pasted") into the net, along the same interface.*

For a more formal definition and a more detailed study of the intricacies of IN interaction, see (Banach, 1995).

The restrictions of allowing only a single edge per port, and preserving the interface in an interaction, have a profound effect on IN. They make INs *linear*, in the sense that variables (free ports) are never created, duplicated or destroyed. In that IN are very similar to LL Proof Nets, of which they are a generalization. This is not the case for most classical formalisms, for example the λ-calculus term $\lambda xy.xx$ duplicates (makes two references) to the argument variable x and erases the argument variable y. If we need to duplicate and erase in IN, we have to use explicit duplicator δ and eraser ϵ nodes. We discuss all this in detail in §2.2.

In order to keep IN reduction deterministic, the following restrictions are imposed on the interaction rules of any IN system.

Definition 2.1.0.4 (Unique Applicable Rule) *No two different rules should have the same LHS.*

This means that for any given cut, there can be only one[2] applicable rule. However, there still might be more than one *way* to apply that single rule. The following takes away that possibility.

Definition 2.1.0.5 (Reflexive Rules Are Symmetric) *If the LHS of a rule is symmetric (the two cut nodes have the same type), then the RHS should also be symmetric.*

A rule with a symmetric LHS is sometimes called *reflexive* because two nodes of the same type are interacting. We can restate the above definition shortly this way: asymmetric reflexive rules are not allowed.

Because pasting the RHS in place of the LHS preserves the interface of the LHS, a closed net will remain closed after reduction. And since one can obtain richer theoretical results for closed nets than for open nets, this is the reason why IN investigations most often consider only closed nets to be "proper".

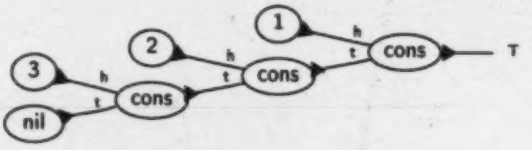
The applicability and effect of a rule depend only on the two nodes that are involved, but not on the context (their other connections). Therefore IN rewriting is purely *local*.

The computation is also *asynchronous* in the sense that there is no global clock (unlike *e.g.* cellular automata). Locality of interaction, asynchronicity and non-interference between redexes (§2.3.1) make IN reduction especially suitable for parallel implementation.

The local interaction between nodes is *binary* and *synchronous* (sequential), in the sense that they both should be present at the time of interaction, and should "present" their principal ports to each other. One can model asynchronous local interaction by introducing intermediary nodes to hold a "message" until the receiver is available.

2.1.1 Example: Simple List Processing

As an example, in this section we represent *lists* and some simple list processing operations in IN. Same as in Lisp, a list is represented as a chain of cons nodes, terminated with a nil node. [⋈ 2.2]

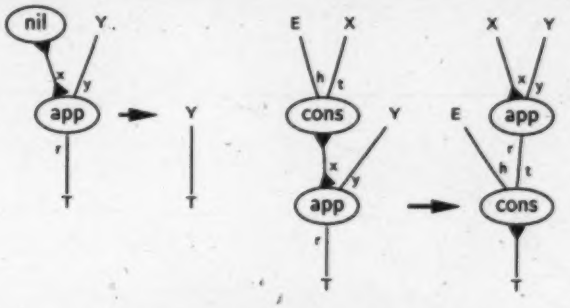


⋈ 2.2: A Sample List, (1, 2, 3).

[2] Or zero.

nil is a zeroary (constant) node with only a principal port. cons is a binary *constructor* node with two auxiliary ports: *head* port h which holds the first element of a list, and *tail* port t which holds the rest of the list.

We now introduce a node app which implements the *append* operation. [⋈ 2.3]



⋈ 2.3: The List "Append" Operation.

In order to understand the above, the most important thing to know is that principal ports do not necessarily correspond to "outputs" of nodes. For "constructor" nodes like cons and nil that is the case, but not for "destructor" nodes (operations) like app. For app, we have indicated the result port with r, and the two input ports with x and y. app is *active* towards one of its inputs only (x).

For a better understanding of the rules, we will explain the letters that we have used (*e.g.* for the LHS of the second rule): T stands for the whole *term* app(cons(E, X), Y). E stands for the first *element* of the first argument list, which is connected to the head port cons.h. The rest X of the first argument list is connected to the tail port cons.t. We can represent the rule in textual form like this:

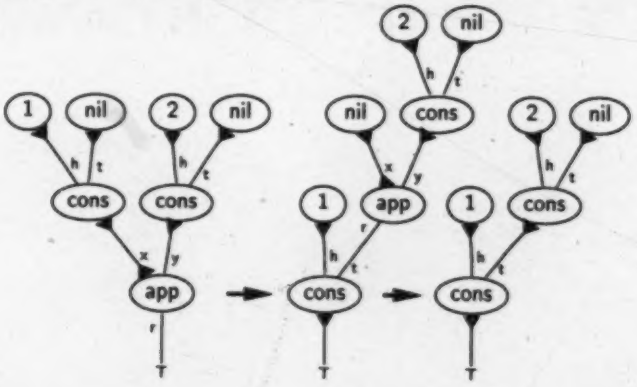
$$\mathsf{T} = \mathsf{app}(\mathsf{cons}(\mathsf{E},\mathsf{X}),\mathsf{Y}) \quad \rightarrow \quad \mathsf{T} = \mathsf{cons}(\mathsf{E},\mathsf{app}(\mathsf{X},\mathsf{Y})),$$

and can restate it in words like this:

> To append a list with head E and tail X to another list Y, first append the two lists X and Y, and then prepend the element E to the front of the resulting list.

Our textual recipe is slightly imprecise, in that it talks about doing something "first" and doing another thing "second". In fact the first element E of the resulting list can be used by T right after the rule is applied; T does not have to wait until the rest of the append operation is completed. This underlines the possibilities of parallel implementation in interaction nets. In the later section §2.5 on Infinite Streams, the possibility for the early use of a partial result (laziness) becomes vital.

Here is how reduction works. For example, the reduction corresponding to appending the lists (1) and (2) to obtain the list (1, 2) is: [⋈ 2.4]

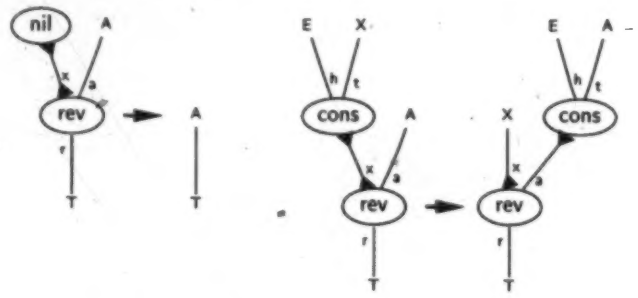


⋈ 2.4: Appending the Two Lists (1) and (2).

As a further example, we introduce a *reverse* operation rev. It is defined using an *accumulator* approach, where the partial result of the reversion is held on the rev.a port. The specification of our reverse operation is:

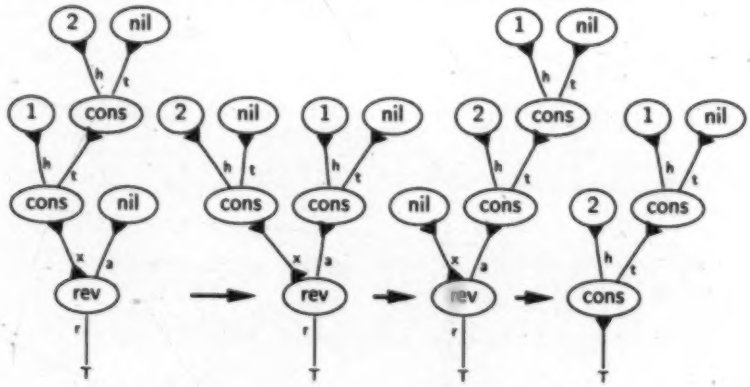
$$\mathsf{T} = \mathsf{rev}(\mathsf{X}, \mathsf{A}) \quad \rightarrow \quad \mathsf{T} = \mathit{append}(\mathit{reverse}(\mathsf{X}), \mathsf{A}).$$

We define the operation like this: [⋈ 2.5]



⋈ 2.5: List "Reverse" Using an Accumulator.

For example, here is how we compute the reversion of the list (1, 2). We invoke the reverse operation giving it nil as a second argument: [⋈ 2.6]

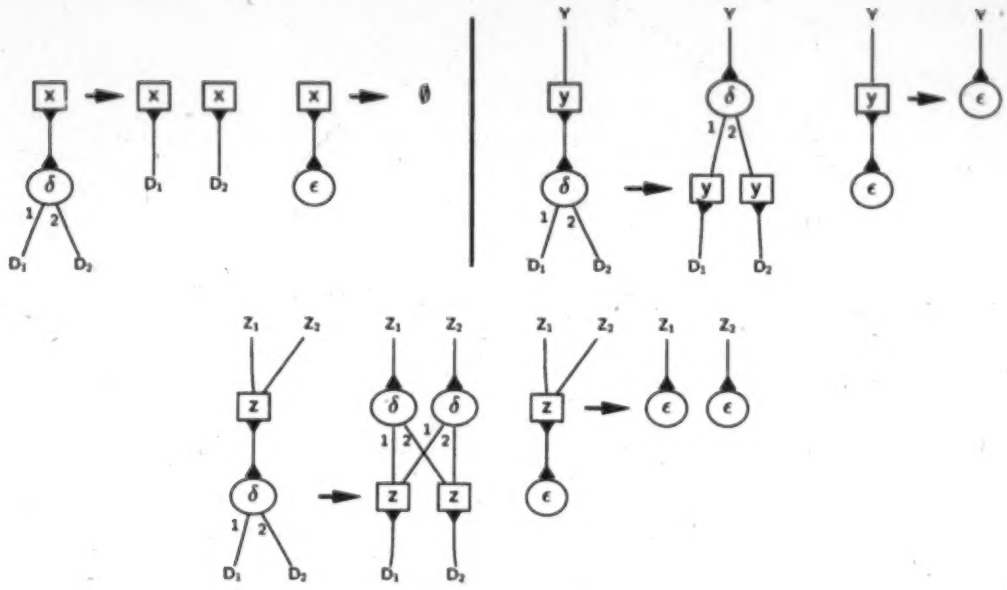


⋈ 2.6: Reversing the List (1, 2).

2.2 Linearity; Duplicator and Eraser Nodes

IN rules are *linear* in the sense that they do not merge nor do they drop variables (free edges). Therefore, if we want to use a piece of net ("datum") twice, or if we want to discard a datum, we need to represent this explicitly. Such explicit representation should not be a shock to anyone acquainted with Linear Logic (LL). In fact, this constitutes a large part of the essence of LL: giving an explicit account of the duplication and erasure of objects (in the case of LL, logical formulae).

For this purpose, we introduce the two nodes *duplicator* δ and *eraser* ϵ. These nodes are "generic", in the sense that they interact essentially the same way with all other node types. Below we present their interaction with "abstract" node types x (zeroary), y (unary) and z (binary); the general idea should be clear. [⋈ 2.7]

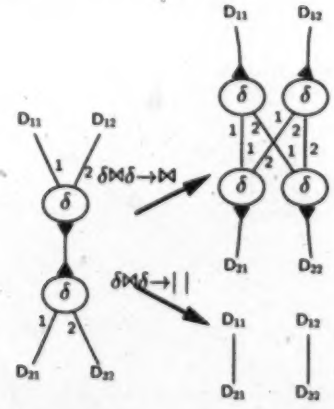


⋈ 2.7: Nodes Duplicator δ and Eraser ϵ.

Looking at the $\delta \bowtie z$ rule, we see that for nodes with $n \geq 2$ auxiliary ports, we need to create n copies of δ on the RHS, so that they can duplicate the net(s) attached to the auxiliary ports of z.

In the rules above, x, y, z stand for any node types except δ and ϵ. It is not quite obvious how to define the interactions of δ and ϵ between themselves:

- We will leave the $\delta \bowtie \epsilon$ interaction undefined.
- There is only one way to define the $\epsilon \bowtie \epsilon$ interaction, namely $\epsilon \bowtie \epsilon \rightarrow \emptyset$.
- There are two reasonable ways to define the $\delta \bowtie \delta$ interaction, and in fact both are used in different systems. We call them the *expanding* or *commutation* rule $\delta\bowtie\delta\rightarrow\bowtie$, and the *short-circuiting* or *annihilation* rule $\delta\bowtie\delta\rightarrow||$. [⋈ 2.8]



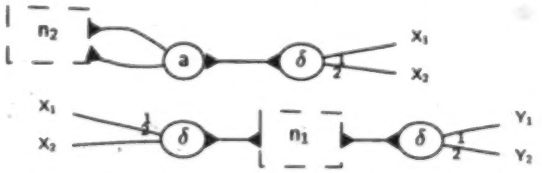
⋈ 2.8: Possible Reflexive δ Interactions: $\delta\bowtie\delta\rightarrow||$ and $\delta\bowtie\delta\rightarrow\bowtie$.

The criterion of which is the correct $\delta \bowtie \delta$ rule to apply is:

- If the two δ's are making two independent copies of a given subnet, then the expanding rule should be used, so that the two δ's "cross" each other safely, and each goes about their copying business.
- If the two δ's are part of the same copying process, in which a net has been "attacked" from several sides, then the event of their meeting each other indicates that they have completed their part of the copying process, and they should retire.[3]

Of course, we would not want to build such a non-local criterion in our IN systems. Rather, we should either 1) ensure that $\delta \bowtie \delta$ never occurs, or 2) adopt one of the two rules and ensure externally that it is the right one to apply, or 3) introduce some auxiliary local machinery ("indexes" on δ nodes) to capture the global strategy. (This is the approach adopted in the theory of Optimal λ-calculus, see (Gonthier *et al.*, 1992b).)

For example, below the two δ's working on the net n_1 should pass each other (use the expansion rule) when they meet in the "middle" of n_1. However, the two δ's resulting from the interaction $a \bowtie \delta$ should annihilate each other when they meet again in the middle of n_2. [⋈ 2.9]



⋈ 2.9: The Kind of the Reflexive δ Interaction Should Depend on the Reduction History.

We give an example of the need for different $\delta \bowtie \delta$ rules in §2.4.2.

2.2.1 Example: Unary Arithmetics

The following example from Lafont (1990) implements the operations plus and times for natural numbers in *unary* notation. A natural number is represented as a chain of *successor* nodes s, terminated by a *zero* node 0. [⋈ 2.10]

⋈ 2.10: The Number 5 in Unary Notation.

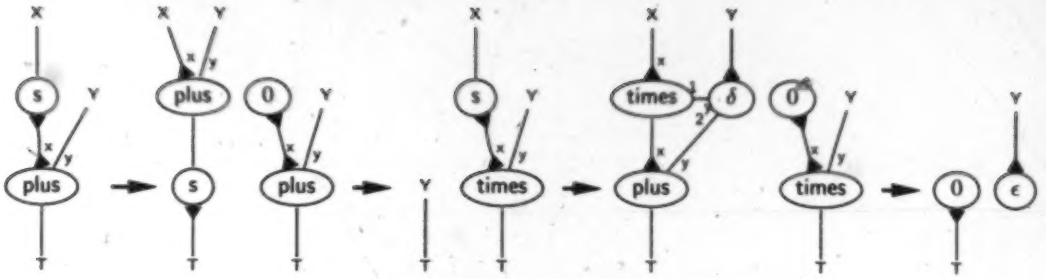
The IN representation is based on the usual definition of + and × in the theory of recursive functions:

$$sx + y = s(x + y) \qquad 0 + y = y \qquad sx \times y = x \times y + y \qquad 0 \times y = 0.$$

Since the recursion "operates" on the argument x of both + and ×, this argument should be assigned to the principal port of the plus and times nodes, and not the output as one might be tempted to do.

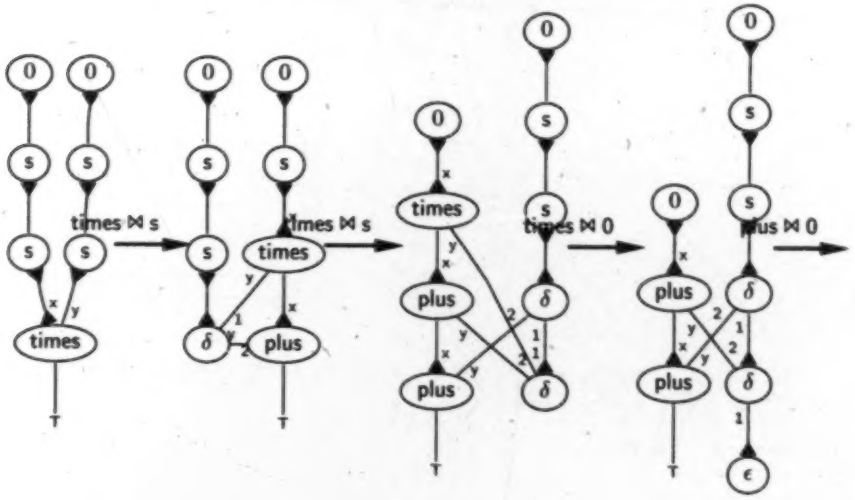
While the equations for + are "linear" in that all variables are present exactly once on each side of the equation, the first equation for × uses y twice (on the RHS), while the second one does not use y at all. Therefore, for the interaction rules involving times, we need to use the duplicator and eraser nodes: [⋈ 2.11]

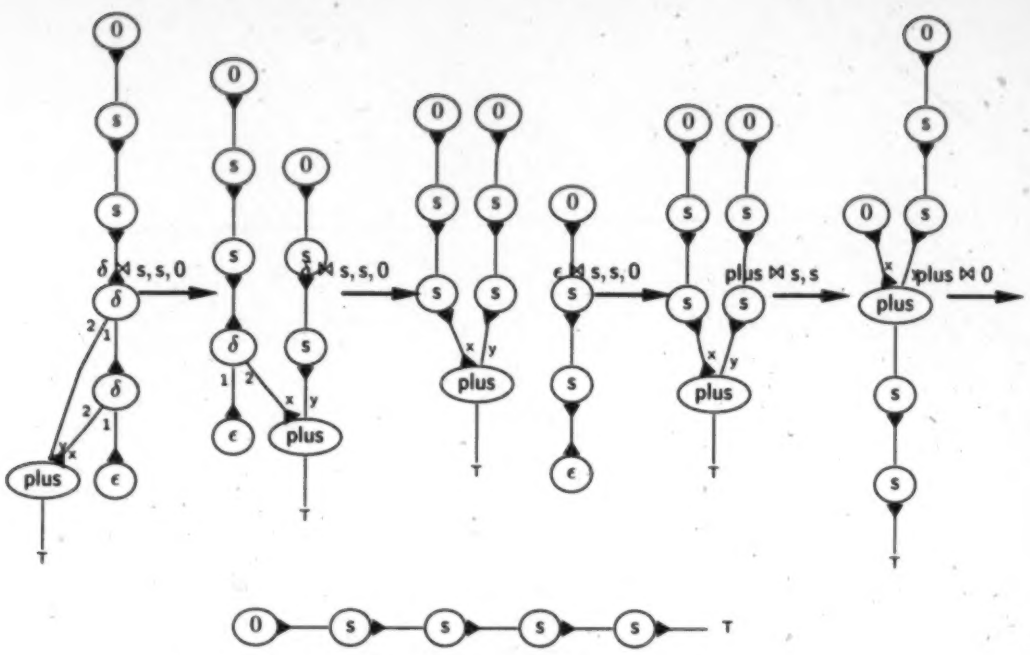
[3] Here is a more graphical example: if two crews are digging two parallel tunnels under a mountain starting from opposite sides, then it's ok for them to pass each other in the middle, without meeting. But if the goal is to dig only one tunnel, then when they meet, their work is over.



⋈ 2.11: Unary Arithmetics Represented in IN.

To verify our definition, we trace the calculation of $2 \times 2 = 4$. [⋈ 2.12]





⋈ 2.12: Even $2 \times 2 = 4$ is Not Trivial in IN.

2.3 Useful Properties of Interaction Nets

Interaction Nets are a restricted form of graph rewriting that enjoys several nice properties:

Binary Interaction Only two nodes participate in a rewriting step.

Locality The applicability of a rule is determined by local inspection, and the effect of a rule is a local modification of the net.

Simplicity It is easy to apply a rule, because its applicability does not depend on the context, and because its effect is easy to compute. The former should be credited to locality, and the latter to linearity (the fact that ports are connected only in pairs), and to the fact that the RHS is pasted along the interface of the LHS.

Confluence and Parallelism The final result of IN reduction does not depend on the order in which we apply rules, and even the length of a reduction does not depend on this order. This leaves us free to implement reduction in any order, or even in parallel.

Blocking/Deadlock-Freeness It is easy to understand essential concurrency concepts like blocking and deadlock, and it is possible to identify syntactically classes of IN which are free of such undesirable phenomena.

In the rest of this section we present some of these properties formally.

2.3.1 Confluence and Determinism

IN rewriting is strongly confluent, because there are no critical pairs. Every two redexes (cuts) are necessarily disjoint, and reductions starting from a redex cannot influence another existing redex.

Therefore, the order in which rules are applied does not matter and in fact the reduction can be done in parallel.

Proposition 2.3.1.1 *INs enjoy a strong form of confluence, the diamond property in single-step reductions. If a net N reduces in one step to a net N_1 by the rule R_1, and N also reduces in one step to a different net N_2 by the rule R_2, then there is a common net Z to which both of N_1 and N_2 reduce in one step:*

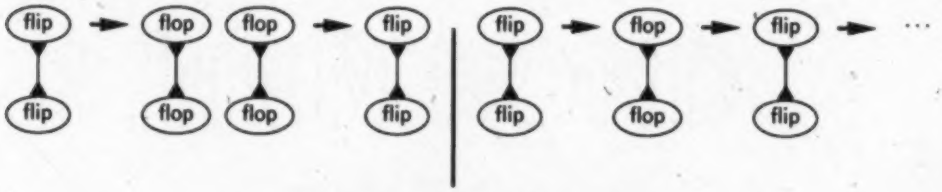
$$\begin{array}{ccc} \forall N & \xrightarrow{R_1} & \forall N_1 \\ {\scriptstyle R_2}\downarrow & & \downarrow{\scriptstyle R_2} \\ \forall N_2 & \xrightarrow{R_1} & \exists Z \end{array}$$

Proof A redex consists of only two nodes connected by their principal ports (2.1.0.2) and the connecting edge. A node has only one principal port (2.1.0.1), so it can participate in at most one redex. Therefore, two different redexes are necessarily disjoint. Since a redex can be reduced by a rule in only one way (see 2.1.0.4 and 2.1.0.5), R_1 and R_2 must have been applied to two different redexes in N. Let $x_1 \bowtie y_1$ and $x_2 \bowtie y_2$ be these two redexes. Then R_1 cannot change $x_2 \bowtie y_2$, and R_2 cannot change $x_1 \bowtie y_1$, therefore they continue to exist in N_1 and N_2 respectively. This allows us to apply R_2 to $x_2 \bowtie y_2$ in N_1, and R_1 to $x_1 \bowtie y_1$ in N_2. In both cases the result must be the same net Z, because of the local way in which rule application acts (2.1.0.3). ◊

There are no critical pairs, and the reduction of a given redex can never preclude the reduction of another redex, therefore the two reductions can be done in either order, or in parallel.

A (closed) net with no cuts is called *reduced*, or a *normal form*. A reduction is called *normalizing* if it leads to a normal form. If a net has one normalizing reduction, than all its reductions are normalizing, and they lead to the same result. Furthermore, all reductions have the same length. Therefore, every normalizable net can reduce in essentially only one way.

This said, IN are still powerful and interesting enough, for example they can model infinite (diverging) computations and the λ-calculus (§2.4.2). Below is a simple example of an infinite computation, a *flip-flop*: [⋈ 2.13]



⋈ 2.13: A Simple Infinite Computation: flip-flop.

2.3.2 Undesirable Situations: Blocking and Deadlock

The following two configurations are undesirable in a net:

Block A cut for which there is no interaction rule defined.

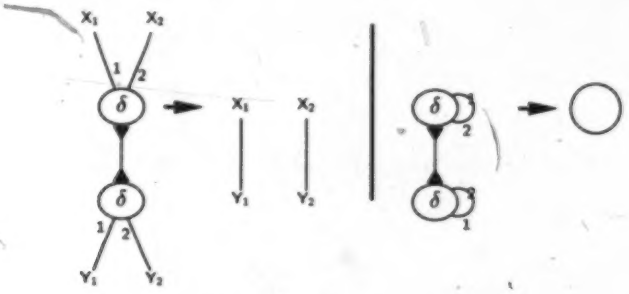
Vicious Circle (also called *deadlock* or *principal cycle*). A cyclic arrangement of nodes such that the principal port of each one is connected to an auxiliary port of the next one.

The block and vicious circle are irreducible: the nodes involved have their interaction capabilities (their principal ports) engaged. But a normal form (fully reduced net) is also irreducible, so why do we call these two "undesirable"? The reason is that the block and vicious circle can *never* become

reducible, even if they are considered in the context of a bigger net. This is not the case of a normal form that is free of blocks and vicious circles. It can be seen easily that such a normal form, if non-empty, necessarily has a free principal port. (See §2.3.4 for background: following a chain of edges $n_1.p \to n_2.a$ from principal to auxiliary ports, we will either hit another principal port (a block), loop (vicious circle), or come to a free port. The former two alternatives are disallowed.) Therefore if we place the normal form in an appropriate context, it will be able to participate in further reductions.

Thus we can say that the interaction capabilities of nodes involved in a block or a vicious circle are tied up *unproductively*, and we should always try to design systems that are free of such phenomena, or more aptly called such defects.

Aside IN reduction can lead to the creation of a cyclical wire, which is an edge closing on itself. For example, given the rule on the left, the initial configuration on the right creates a cyclical wire. [⋈ 2.14]



⋈ 2.14: Creation of a Cyclical Wire.

Here the cycle is created as a result of the two edges connecting ports on the same node, but we don't want to disallow such edges, because they are useful sometimes. Anyway, cycles can be created as a result of more complicated node arrangements as well. Depending on the situation, cyclic wires are regarded sometimes as harmless garbage, and other times as vicious circles.

Blocking can be avoided by introducing a suitable type system. Ports are annotated with types, and only same-typed[4] ports can be connected by wires. The interaction rules should be well-typed (respect the typing of nodes, and preserve the typing of the interface), and it is easy to see that then reduction also respects the typing (*subject reduction*). In fact the distinction between principal and auxiliary ports is sometimes regarded as a simple typing discipline of its own. We will not study type disciplines in any detail here, instead we refer the reader to Lafont (1990), Fernández (1998), and the extensive literature on types for the λ-calculus and the π-calculus. Instead, we focus on the second problem.

2.3.3 Deadlock Avoidance

The general problem of determining whether a given IN system is deadlock-free (guaranteed to never produce a principal cycle from an initial net that doesn't have one) is undecidable. Therefore, often we have to be content with *sufficient* (but not necessary) conditions for deadlock-freeness, conditions which allow us to verify nets falling in a particular class. Such conditions are stronger than deadlock-freeness (imply deadlock-freeness), and are easier to maintain (it is easier to prove that they are preserved during reduction). We present here two such conditions.

The main reason of Lafont to limit his consideration to INs with single principal ports is that it is easier to study deadlock-freeness criteria for that case. Otherwise one is allowed greater freedom (a principal cycle can be broken if a node involved in the cycle has another principal port), but

[4] Or oppositely-typed, depending on the system.

deadlock-freeness criteria are more complex. (Another reason is that linear logic Proof Nets are naturally single-port INs.) Banach (1995) considers nodes with more than one principal port and gives some deadlock-freeness criteria for such nets. Banach also studies generalized criteria similar to simplicity for the case of single principal ports.

Partitions This notion was introduced by Lafont (1990).

Let's assume that for a given IN system S and for every node type $n \in S$, a *partition*[5] of the auxiliary ports of n is given. Then a net is called *simple* if it can be built by structural induction using the following rules:

- A single edge is a simple net.
- Two simple nets connected by a single edge form a simple net.
- Given a node n with partitions $\{p_1 \ldots p_k\}$ and k nets $N_1 \ldots N_k$, the net N formed by connecting each N_i to the partition p_i is a simple net. (This operation is called *Grafting* the nets N_i onto the node n.)

It is easy to establish the following:

Fact 2.3.3.1 *A simple net cannot contain a principal cycle.*

Proof By structural induction. We only need to check the Graft operation. Assuming that $N_1 \ldots N_k$ are free of principal cycles, an alleged principal cycle in N must pass through n. However, Graft leaves n's principal port free, therefore it cannot partake in any principal cycle. ◊

Please note that the criterion of simplicity is quite strong. For example, the translations of λ-calculus terms on ⋈ 2.20 are not simple nets:

- By the middle construct, the two auxiliary ports of λ must be in the same partition (to allow the graft of $[b]$ to λ).
- By the right construct, the two auxiliary ports of λ must be in different partitions, to allow the graft of two unconnected nets ϵ and $[b]$ to λ.

An interaction rule is called *simple* if its RHS is a simple net, where the interface of the RHS is partitioned according to the partitions of the LHS.

Theorem 2.3.3.2 *If a simple rule $L \to R$ is applied to a simple net M, the result is a simple net N. Thus reduction by simple rules preserves simplicity of nets.*

Proof Similar to a proof of Cut Elimination in Linear Logic; we only sketch the proof here. Let $L = m \bowtie n$. By some "permutations" of inductive formation rules, we can assume that the last three steps in the construction of the simple net M are: 1) Graft some nets onto m, 2) Graft some nets onto n, 3) Add the cut edge $m \bowtie n$. Now we can repeat the same construction for the net N, replacing the last three steps with the construction showing that R is a simple net, This gives us an overall construction showing the simplicity of N. ◊

Switchings This notion was introduced by Lafont (1991), following ideas from Linear Logic Proof Nets (Danos and Regnier, 1989).

A *switch* assignment of a node occurrence n is a pairing of some of the ports of n (think of it as wiring inside the node): $s_n = \{n.p_i - n.p_j, \ldots\}$. The *switching* S_n of a node type n is defined as the set of allowed switch assignments for that node type. We impose the following condition on node switchings:

[5] An exhaustive non-intersecting set of subsets.

Condition 2.3.3.3 (Principality of Switchings) *For every node* n *and each one of its auxiliary ports* $\mathsf{n.a}$*, there must be a switch* $s \in S_{\mathsf{n}}$ *connecting* a *to the principal port* $\mathsf{n.p}$ *of* n*:* $(\mathsf{n.a}{-}\mathsf{n.p}) \in s$.

A *switching* of a net N is a set of switch assignments $\{s_{n_1} \ldots\}$ for the nodes of N that respects the switchings of the nodes: $s_{n_i} \in S_{\mathsf{n}}$ where n is the type of the occurrence n_i. Different occurrences of the same node type are allowed to have different switch assignments: $s_{n_i} \neq s_{n_j}$. We call a net with a particular switching a *switched net*.

A *switched cycle* is a sequence

$$(\mathsf{n}_1.\mathsf{p}_1{-}\mathsf{n}_2.\mathsf{q}_2, \mathsf{n}_2.\mathsf{p}_2{-}\mathsf{n}_3.\mathsf{q}_3, \ldots \mathsf{n}_k.\mathsf{p}_k{-}\mathsf{n}_1.\mathsf{q}_1)$$

of edges in a switched net, such that $(\mathsf{n}_i.\mathsf{q}_i{-}\mathsf{n}_i.\mathsf{p}_i) \in s_{n_i}$. If we follow the intuition above that switches are wirings inside a node ("internal edges"), then a switched cycle is a cycle formed by alternating internal and external (normal) edges. We call a net *correct* if it is free of switched cycles, *i.e.* if no allowed switching of the net creates a switched cycle.

An immediate consequence of our definitions and Condition 2.3.3.3 is

Fact 2.3.3.4 *For every principal cycle in a net, there exists a switching which produces a switched cycle.* ◊

Therefore, a correct net is deadlock-free.

Let's recount what we have done until now: a principal cycle is a cycle formed of external (normal) edges and internal edges connecting an auxiliary port to a principal port. We are enriching the set of allowed internal edges (we permit some auxiliary-to-auxiliary internal edges as well), and still require our nets to be cycle-free. In other words, we are introducing some redundancy in the allowed internal edges and are making our nets more "cycle-repulsive", so that it is easier to maintain this criterion of cycle-freeness during reduction.

Now we want to impose a condition on interaction rules so that switched acyclicity is preserved by reduction. Consider a rule $\rho = L \to R$ and its application in a net $M \xrightarrow{\rho} N$. Let $C = M - L = N - R$ be the context in which the rule is applied, and I be the interface (the set of free ports of C and L and R; these three sets coincide). We want ρ to be such that every switched cycle in N can be attributed back to a switched cycle in M. It is enough to ensure this for the simplest possible kinds of C, namely *wirings*, *i.e.* disjoint pairings of ports in I.

Definition 2.3.3.5 (Legal rules) *A rule* ρ *is called* legal *if for every wiring* W *of the interface* I *and switching* s_R *of the RHS that create a switched cycle in the net* $W + R$*, there exists a switching* s_L *of the LHS that creates a switched cycle in the net* $W + L$.

We can now prove the following

Proposition 2.3.3.6 *If* M *is a correct net and* $\rho = L \to R$ *is a legal rule and* $M \xrightarrow{\rho} N$*, then* N *is a correct net.*

Proof Assume to the contrary that there is a switching s_N that creates a switched cycle in N. If that cycle is entirely in $C = N - R$, then it immediately gives us a cycle in $M = C + L$. Else if the cycle is entirely in R, then since ρ is legal, Definition 2.3.3.5 instantiated with $W = \emptyset$ gives us a cycle in M that is entirely in L. Otherwise, the cycle consists of two parts, one inside C and the other inside R. Similarly, we may consider s_N to consist of two parts, s_C and s_R. C and s_C determine a wiring W of the interface ports, namely two ports are paired in W if there is a switched path that connects them in C with switching s_C. Apply 2.3.3.5 to obtain a switching s_L that together with W forms a switched cycle in $L + W$. This cycle corresponds directly to a switched cycle in $M = L + C$, namely the one created by s_L and s_C. ◊

Checking the Deadlock-Freeness of Arithmetics As an example, let's go back to our rules for unary arithmetics ⋈ 2.11 and ⋈ 2.7 and prove that they are deadlock-free. We will use the second method, *i.e.* define node switchings that would render the rules legal. If we succeed in finding such switchings, this will constitute a proof that unary arithmetic nets are correct, and thus deadlock-free:

1. Consider the rule plus ⋈ 0. Since a wiring Y−T on the RHS creates a cycle, we need to "prevent" this cycle by creating a corresponding cycle on the LHS. For this, we need the switch plus.t−plus.y. By condition 2.3.3.3, plus also allows the other two switches plus.t−plus.x and plus.y−plus.x, since x is its principal port. Therefore, we can call the plus node a "3-switch", since it allows all 3 possible switches. There are no other possible cycles on the RHS, so we are done with this rule (it is now legal).

2. Now consider the plus ⋈ s rule. plus being a 3-switch and s being necessarily a 1-switch, every wiring of the RHS leads to a cycle. Luckily, we can recreate all these cycles on the LHS. For example, consider the wiring X−Y. The LHS switched cycle involving this wiring is X−s−plus.x−plus.y−Y−X. This rule is legal.

3. The times ⋈ 0 rule is trivially legal, since there can be no RHS cycles, no matter what wiring we consider.

4. Now consider the times ⋈ s rule. plus being a 3-switch by 1, the Y−T wiring on the RHS leads to a cycle: Y−δ.2−plus.y−plus.t−T. In order to "prevent" this cycle, we need a corresponding cycle on the LHS, therefore we need to allow the switch times.y−times.t. Thus times must also be a 3-switch. From this follows that the switch δ.1−δ.2 must *not* be allowed, or else the RHS will permit a cycle contained entirely within itself:

   $$\delta.2-\text{plus.y}-\text{plus.x}-\text{times.t}-\text{times.y}-\delta.1-\delta.2$$

   Therefore δ must be a 2-switch but not a 3-switch. The other possible wirings X−Y and X−T create principal switched cycles on the LHS, so they are not dangerous. This rule is legal.

5. Now we must check the rules for δ and ϵ. We start with ⋈ 2.7. All ϵ rules are trivial, as is the δ ⋈ x rule (where x is zeroary). The unary rule δ ⋈ y is also easy: the D_1−Y and D_2−Y wirings create a cycle on the LHS, while the D_1−D_2 wiring does not create a cycle on the RHS, since δ.1−δ.2 is not a permitted switch. The binary δ ⋈ z rule is most complicated, since there are 10 possible wirings. We will assume that z is a 3-switch (the other case of z being a 2-switch is considered below for the rule δ ⋈ δ.

   - If Z_1−Z_2 is in the wiring, this creates a LHS cycle Z_1−z−Z_2.
   - If Z_i−D_j is the wiring $(i, j = 1, 2)$, this creates a principal switched cycle on the LHS.
   - If the wiring is {D_1−D_2}, there can be no RHS cycle since δ.1−δ.2 is not a permitted switch.
   - Finally, if the wiring is empty, there can be no RHS cycle for the same reason.

6. Consider the ⋈ 2.8 rule δ⋈δ→⋈. We only need to check the case when D_{11}−D_{12} is in the wiring, since the other cases go just like in the previous item. Unfortunately, the wiring {D_{11}−D_{12}, D_{21}−D_{22}} leads to a RHS cycle that does not have a corresponding LHS cycle (a cycle cannot use the cut edge δ ⋈ δ on the LHS twice). The same problem arises if we choose the rule δ⋈δ→| |. No other cycles present a problem.

From the last item above, it follows that our arithmetics rules are not correct, unless we banish the δ ⋈ δ rule from our system. For this measure to be acceptable, it is necessary to check that such a redex will never occur, else we'll end up with a block.

A simple inspection of the ⋈ 2.11 and ⋈ 2.7 rule sets convinces us that indeed the δ ⋈ δ rule can be banished safely: ⋈ 2.11 introduces only δ's that are oriented "upwards" (with their port away from the root of the term T), and ⋈ 2.7 doesn't change the orientation of a δ.

2.3.4 Structural Result: "Combing" a Deadlock-Free Net

A net is a tangled mass of edges going in all directions. Let's try to make more sense of this mass by "organizing" the edges into a system that's easier to understand. For graphical effect, we will call this construction "combing a net".

Let's consider the set of edges of a net. They are of three types:

- Cut $n_1.p_1 \bowtie n_2.p_2$ between two principal ports.
- Oriented arc $n_1.p_1 \rightarrow n_2.a_2$ from a principal port p_1 to an auxiliary port a_2.
- Unoriented edge $n_1.a_1 - n_2.a_2$ between two auxiliary ports a_1 and a_2.

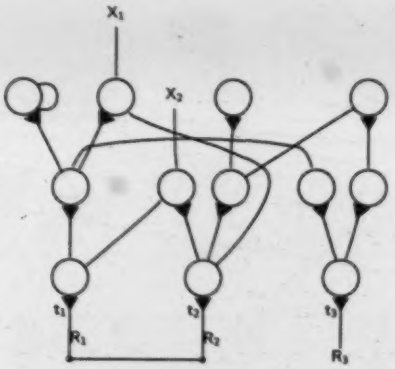
There may also be free edges, whose second end is not connected to any port. We will assume the missing port to be "auxiliary", so we will consider such edges to be of the latter two types.

Now let's consider a net N without vicious circle (cycle of oriented arcs, or a principal cycle). First let's split all cuts of the net and call the resulting free principal ports, together with the original free principal ports of the net, the *roots*. We also get a set of arcs attached to, and pointing towards, the roots.

Then lets "comb" all oriented arcs in the same direction —towards the roots— using the following reasoning. Consider one of the arcs. It enters its target node from an auxiliary port. Since the node has only one principal port, there is only one way to exit the node following an arc. Continuing this process, we will reach a root, since there are no oriented cycles. This gives us a forest (set of oriented trees) rooted at the roots, and the forest covers all nodes, since every node has an outgoing arc (principal port).

What is left of the net are unoriented edges. These edges connect some of the auxiliary ports of the trees together (others are free ports of the net). We leave these edges "uncombed".

To summarize, we have a set of roots, some of which are connected by cuts, oriented trees rooted at these roots, and unoriented edges connecting auxiliary ports of the trees. [⋈ 2.15]



⋈ 2.15: Typical "Combed" Net.

Note A reviewer kindly pointed that this amounts to the construction *topological sorting*, well-known from graph theory.

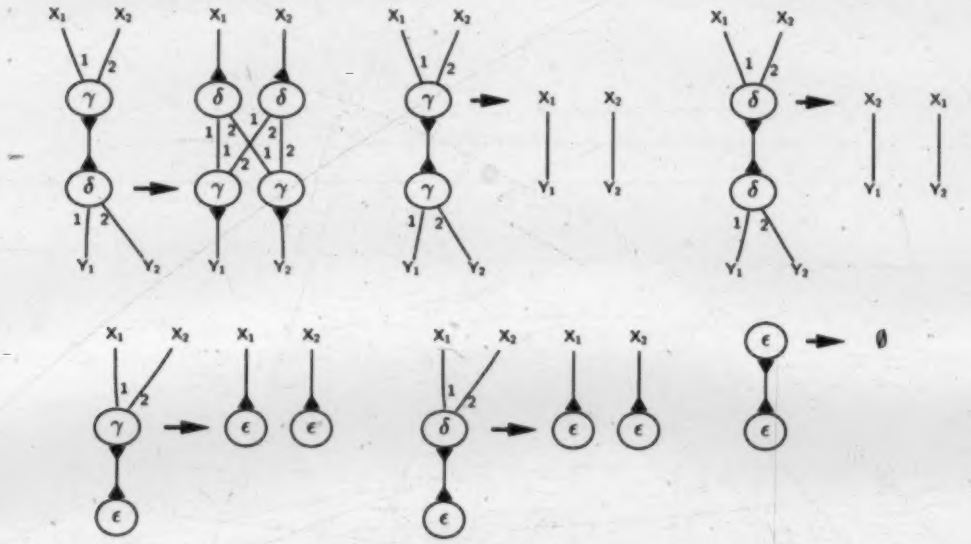
2.3.5 Combinatorial Completeness of IN

We will see later (§2.4.2) that IN are computationally complete in the sense of Church and Turing (they can represent the λ-calculus). But INs represent a very different paradigm from the λ-calculus, namely parallel computation instead of sequential functional computation. Therefore, it is a very interesting question whether INs are *complete* "within themselves", *i.e.* whether there exists a

universal IN system that can capture any other IN system in a faithful way. By "faithful" we mean that the degree of parallelism of the original system is preserved, for example a number of independent rule applications to disjoint redexes should be translated to a number of independent sub-reductions which may also proceed in parallel.

Lafont (1995b) answers this question in the affirmative, by constructing a system of *interaction combinators* IC that is universal in the above sense. Lafont's system is very simple, it consists of 3 node types (combinators) (2 of which we have already introduced), and 6 interaction rules. On the other hand, the translation of an arbitrary IN system into IC is quite involved. Gay (1994) constructs independently a similar system, which however is more complicated than Lafont's.

The system IC consists of 3 node types: *eraser* ϵ, *duplicator* δ (we have already introduced these two in §2.2), and *constructor* γ. [⋈ 2.16]



⋈ 2.16: The Interaction Combinators of Lafont.

(Please note that the $\delta \bowtie \delta$ reduction is defined slightly differently than any of the alternatives given on ⋈ 2.8: this reduction "twists" the two wires on the RHS and thus acts differently from the $\gamma \bowtie \gamma$ reduction. This is essential for the completeness of the system IC.)

That such a simple system as IC can represent every possible IN system S is a very interesting, surprising, and highly non-obvious result. We only outline it below, and send the interested reader to (Lafont, 1995b) for the details. The basic idea is that every node n of S is represented as an IC net that encodes all possible interactions of n. The nets used in this encoding are *principal*, which means that they have one principal port, and as many auxiliary ports as the nodes they represent. In addition to wiring/connection issues (for which the combinators of IC are perfect), ideas from linear logic are widely used in this translation (duplication, erasure, selection of one alternative amongst many). The following issues arise and are resolved by IC constructions:

Multiplexing/Transposition We will often need to "gather" a bunch of n wires using *multiplexors* M_n and "output" them as one single wire through M_n's principal port, such that when M interacts with a complementary multiplexor M_n^*, the wires are reconnected together. One possible implementation of these multiplexors is as a ladder (chain) of $(n-1)$ γ nodes, which are chained on their right (number 2) ports for M_m, and on their left (number 1) ports for

M_n^*.[6] We can also easily define *transpositors* T_n^σ for any involutive permutation σ of the numbers $\{1 \dots n\}$. (An involutive permutation is a pairing of some of the numbers in the set.) When two transpositors of the same type interact, they connect their free ports according to the permutation σ. One possible implementation is again to use a ladder of binary IC nodes, but in this case we need to use both γ (for ports that are connected "straight through") and δ (for ports that are "swapped"). Once we define these elements, we have enough "wiring machinery" to connect ports in various ways.

Menu/Selection We need to be able to gather several "alternative" nets together into one aggregate net, and later select one of them, while discarding the others. Let $N_1 \dots N_n$ be n principal nets. The *menu* composed of these nets is indicated $N_1 \& \cdots \& N_n$,[7] and can be implemented simply as the multiplexor M_n, with the n nets connected to its auxiliary ports. Then a *selector* S_n^i should fetch the ith alternative from the menu, discarding the others. We can implement it as a complementary multiplexor M_n^*, with erasers ϵ connected to all of its auxiliary ports except the ith one.

RHS Parts As we mentioned, a node n of N will be represented as a menu of all possible interactions in which n can participate. Let

$$\mathsf{m}(X_1 \dots X_k) \bowtie \mathsf{n}(Y_1 \dots Y_l) \to R$$

be an interaction rule involving n (here k and l are the arities of m and n respectively). Then we can split the RHS of the rule in two subnets R_m and R_n, where R_m has the free ports corresponding to the auxiliary ports of m, and R_n has the free ports corresponding to the auxiliary ports of n. (Using the "combing" representation of §2.3.4, we can define a canonical such splitting.) The intent of the splitting is to be able to "attribute" parts of the RHS of a rule to each of the two nodes on the LHS.

Then the translation of n is a menu of all possible RHS parts[8] attributed to it: $[\![\mathsf{n}]\!] \stackrel{\text{def}}{=} R_n^1 \& \cdots \& R_n^T$, where T is the total number of node types of the original system S. If S does not define a particular interaction $\mathsf{m} \bowtie \mathsf{n}$, then we can define R_n^m arbitrarily, for example as a single eraser ϵ.

Coding/Decoder There is a significant defect with the description of $[\![\mathsf{n}]\!]$ so far: it mentions RHS parts R_n^m, but these nets contain nodes x from the original system S, while we must limit ourselves to only nodes from IC (combinators). We cannot simply replace x with $[\![\mathsf{x}]\!]$, because our definition of translation would fall into an infinite recursion, and the translation of a node would expand infinitely. Therefore, in addition to the "real" $[\![\mathsf{n}]\!]$, we also need to somehow encode (freeze) this translation using only IC nodes, and then be able to use it to reconstruct the "real" $[\![\mathsf{n}]\!]$.

Duplication We need the ability to duplicate parts of a net, so that one copy can be used to emulate the reduction of S, and a second copy to regenerate the "frozen" code and carry it into the result of the reduction for future use.[9] Typically we use δ to duplicate nets, but a complication presents itself here: δ cannot duplicate nets containing δ. Lafont (1995b) resolves this problem by introducing a secondary code and a decoder for it, but we will not go into the details.

---

[6] For the special degenerate cases define $M_0 = M_0^* = \epsilon$ and $M_1 = M_1^*$ is a single wire.

[7] The symbol & used in this context comes from the additive-and (*with*) connective of linear logic, which has similar properties.

[8] Plus some wiring elements.

[9] In linear logic, the "freezing" of a logical formula A is denoted $!A$. Duplication is expressed by the logical rule $!A \multimap !A \otimes !A$ (contraction) where $\multimap$ is linear implication, and $X \otimes Y$ (tensor) expresses the simultaneous existence of two formulas (resources) X and Y. "Unfreezing" (the use of a frozen formula) is expressed as $!A \multimap A$ (dereliction), and erasure (discarding of a frozen formula) is expressed as $!A \multimap \mathbf{1}$ (weakening), where $\mathbf{1}$ is the neutral element of $\otimes$. From contraction and dereliction we can also deduce the rule $!A \multimap A \otimes !A$, which is what we need here: the ability to get an unfrozen A for "current operations" (emulation of reduction), and keep a copy $!A$ for future use.

The IC translation of a system S models faithfully the reductions of S, in the sense that for every single-step reduction $M \to N$ in S there is a multi-step reduction $[M] \Rightarrow [N]$ in IC, but it is usually not at all obvious how to interpret the final result. Since the translation $[N]$ is complex, it is not easy to recognize the nodes of N as subnets of $[N]$. This is the classical *read-back* problem of implementation translations (encodings), and we address one such problem in our translation of the finite π-calculus to a modified version of IN (§5.2).

Furthermore, due to the distributed nature of INs, the IC system does not "know" to emulate only the $M \to N$ reduction, and it may interleave this reduction with others. In fact we can not even guarantee that $[M]$ will pass through a state $[N]$ in the course of its evolution. We can only guarantee that every partial reduction $[M] \Rightarrow M'$ can be completed to a reduction $M' \to [P]$, the result of which is the translation of a net in S, and that also $M \Rightarrow P$ holds in S.

2.4 Applications of IN

INs are computationally complete, in that they can capture the λ-calculus and similar applicative systems. To show this, we will start by representing the simplest system of functional combinators SK in IN. By a well-known result of Curry, the SK system can represent the λ-calculus.

2.4.1 SK Combinators

The SK system is defined by the following equations:

$$Sxyz = xz(yz) \qquad Kxy = x.$$

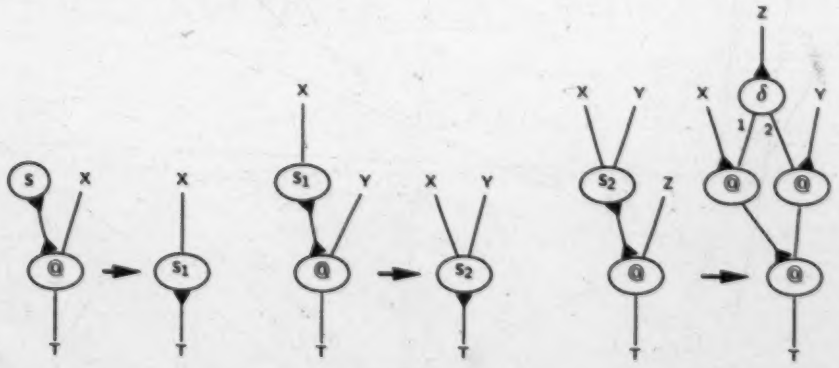
Here juxtaposition expresses functional application, and application associates to the left. In preparation for translating these rules to IN, we first write them in a different form. We introduce an explicit symbol @ for application, denote S and K as s and k respectively, and denote variables with capital letters as we have been doing untill now.

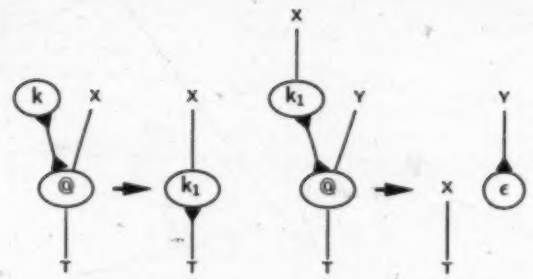
$$@(@(@(\mathsf{s},\mathsf{X}),\mathsf{Y}),\mathsf{Z}) = @(@(\mathsf{X},\mathsf{Z}),@(\mathsf{Y},\mathsf{Z})) \qquad @(@(\mathsf{k},\mathsf{X}),\mathsf{Y}) = \mathsf{X}.$$

We can represent this in IN almost directly. There are only two differences:

- Since IN rules are binary, we can't represent the LHS of the equations above directly, and instead we have to introduce auxiliary nodes $\mathsf{s}_1, \mathsf{s}_2, \mathsf{k}_1$ that accumulate the variable arguments before the final step. Another way to explain this is that INs are *locally sequential*, so local steps have to be sequenced with the use of auxiliary node types.
- We need to represent duplication (in the S rule) and erasure (in the K rule) explicitly.

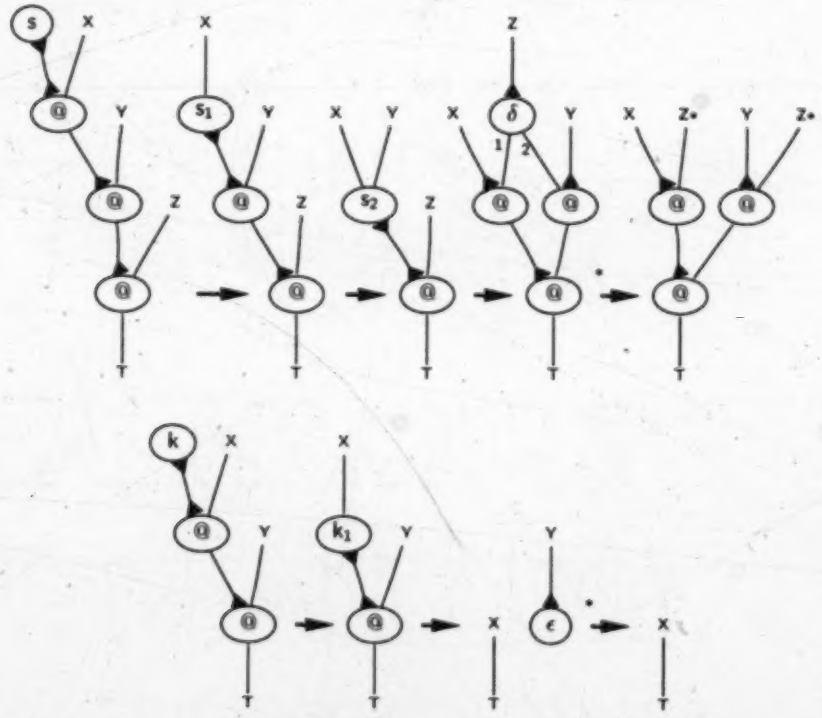
With these remarks, the translation is simple: [⋈ 2.17]





⋈ 2.17: The SK Combinatory System Represented in IN.

Now let's check that our translation indeed implements the SK equations: [⋈ 2.18]



⋈ 2.18: SK Reduction Represented in IN.

The last reductions of the above sequences require some explanation: 1) They are multi-step reductions (that is why they are denoted with a star). 2) Since δ and ϵ can only operate on principal ports, the nets "above" them need to be turned with their principal ports towards δ and ϵ before they can complete their job. Therefore, these nets need to be values (trees of s and k), because @ is turned away from δ and ϵ. We have indicated with Z* above the canonical form of Z (the net resulting from reducing it until it becomes a value).

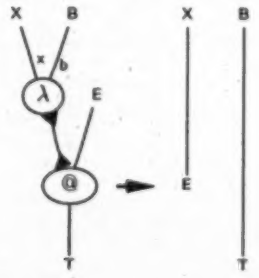
Therefore the above representation is quite non-optimal. It copies every argument Z, even in cases when such an argument will be erased in the very next step. Furthermore, it does not allow sharing; it always copies shared variables apart. On the positive side, the translation is simple, almost obvious.

2.4.2 The λ-calculus

In the previous section we've shown a translation of SK combinators in IN. However, INs are flexible enough to be able to represent the λ-calculus directly as well. INs are also "subtle" in that they can express all aspects of computation in one framework, and thus obtain more realistic estimates about the complexity of a computation. For the λ-calculus, INs model not only β-reduction, but also substitution and the duplication/distribution of values to the several occurrences of a variable.

The representation of the λ-calculus in IN has been part of the LL and IN "lore" for quite a while, and Mackie (1994) gives one of the simplest possible translations.

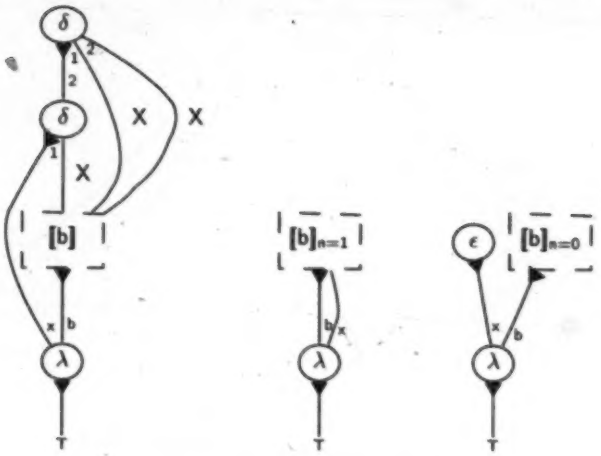
We have already introduced above the ***application*** node @. We now introduce the ***abstraction*** node λ. Given a λ-calculus term B (***body***) on its b port and an abstraction variable X on its x port, the λ node "constructs" and outputs an abstraction, a single-argument function $\mathsf{X} \mapsto \mathsf{B}[\mathsf{X}]$. For simplicity, we first assume that the body B is ***linear*** in X, *i.e.* that X appears exactly once in B. Under that assumption, the substitution of X in B is expressed as a simple short-circuiting: [⋈ 2.19]



⋈ 2.19: β-reduction in the Linear λ-calculus.

If X appears more than once in B, then we need to introduce δ nodes that gather all these occurrences. Conversely, if X does not appear in B, then we need to introduce ϵ nodes to kill the un-needed value. We define the translation $[\![t]\!]$ of a λ-calculus term t as follows. For simplicity, we assume that every bound variable of t has been α-renamed apart from all free variables, and from all other bound variables:

1. If t is a variable x, then $[\![t]\!]$ is a single wire labeled X.
2. If t is the application fe of a function f to an expression e, then $[\![t]\!] = @([\![f]\!], [\![e]\!])$.
3. If t is $\lambda x.b$, then $[\![t]\!]$ is defined as follows: [⋈ 2.20]



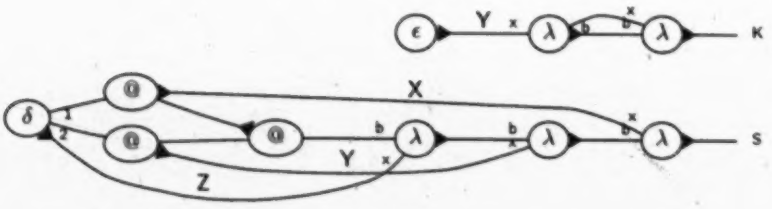
⋈ 2.20: Translation of an Abstraction $\lambda x.b$.

Above λ.b is connected to the single free principal port of the net **[b]** (it is a simple matter to prove that **[b]** has this property). We introduce a chain of $n-1$ duplicators δ, where n is the number of occurrences of x in b, or equivalently the number of free edges of **[b]** labeled with X. We have shown the special cases of $n = 1$ (a simple edge λ.x−**[b]**.x) and $n = 0$ (in which case the variable port λ.x is connected to an eraser ϵ).

As a paradigmatic example, we give the translations of the combinators

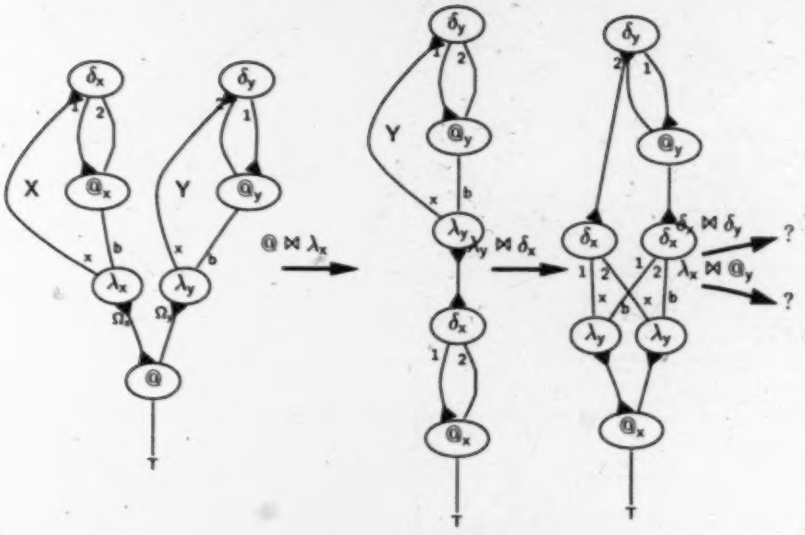
$$S = \lambda x.\lambda y.\lambda z.@(@(x,z),@(y,z)) \qquad K = \lambda x.\lambda y.x$$

[⋈ 2.21]



⋈ 2.21: The Translation of S and K Considered as λ-functions.

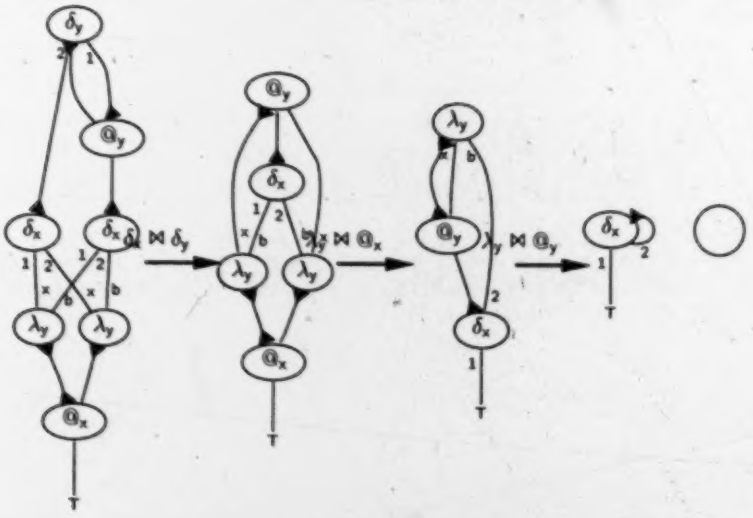
As an example of λ-reduction in IN, we will trace some steps of the reduction of $@(\Omega,\Omega)$, where $\Omega = \lambda x.@(x,x)$. It is well-known (and easy to check) that $@(\Omega,\Omega)$ generates a non-terminating reduction, so this will also serve as an example of infinite reduction in IN. We start with two instances of **[Ω]** connected by an @. In order to be able to distinguish the two **[Ω]** subnets, we define them to correspond to $\Omega_x = \lambda x.@(x,x)$ and $\Omega_y = \lambda y.@(y,y)$ and mark their nodes with subscripts x and y respectively. [⋈ 2.22]



⋈ 2.22: The Reduction of $[@(\Omega,\Omega)]$ (part 1).

At this point we have a choice: do we reduce $\lambda_x \bowtie @_y$, or $\delta_x \bowtie \delta_y$? The general theory of IN tells us that the choice won't make any difference, since IN reduction is confluent (§2.3.1.1).

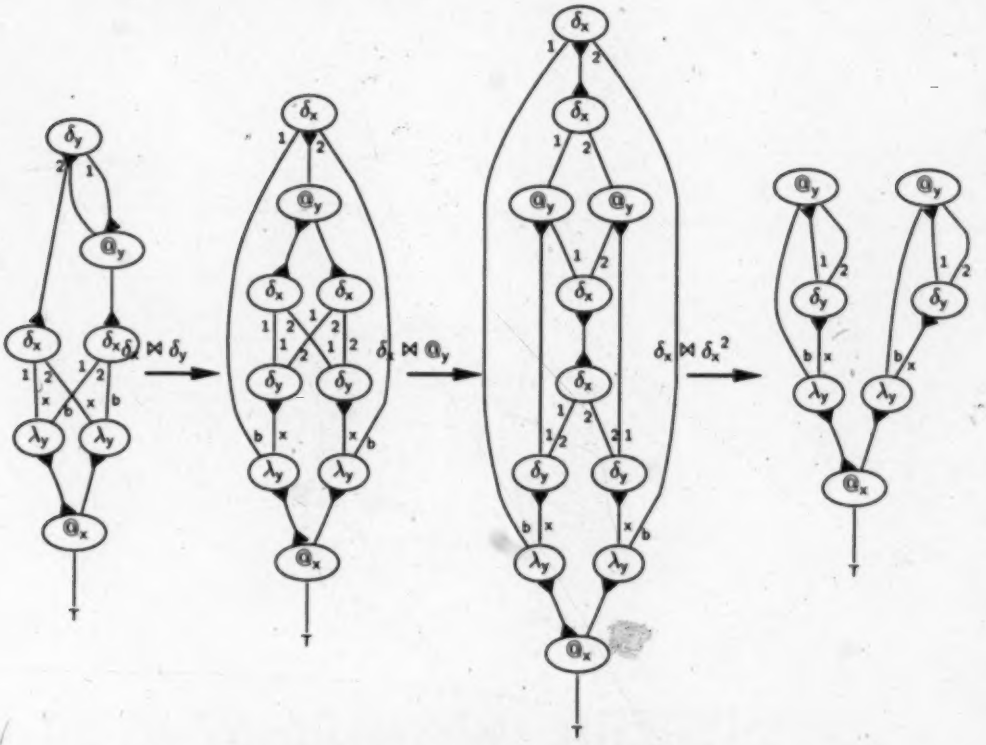
However, if we select $\delta \bowtie \delta$, we also have a real choice to make: do we apply the "short-cutting" rule $\delta\bowtie\delta\rightarrow||$ or the "expanding" rule $\delta\bowtie\delta\rightarrow\bowtie$ of ⋈ 2.8? To answer this question, we should trace back the "genesis" of the two δ's. The two δ's in question were created to duplicate two different variables: δ_x was begotten from x, while δ_y was begotten from y. Therefore, it would be wrong to use the short-cutting rule and thus merge the two variables. Let's see what happens if we do so: [⋈ 2.23]



⋈ 2.23: Applying the Wrong Rule $\delta\bowtie\delta\rightarrow||$.

We obtain a vicious circle (deadlock) pretty quickly. Since the $@(\Omega,\Omega)$ reduction in the λ-calculus is non-terminating, we must have made a mistake.

Now let's make the right decision $\delta\bowtie\delta\rightarrow\bowtie$: [⋈ 2.24]



⋈ 2.24: The Reduction of $[@(\Omega,\Omega)]$ (part 2).

The result is again $@(\Omega,\Omega)$, so we are back where we started. Please note that in the last (two) steps in the reduction above we used the $\delta\bowtie\delta\rightarrow||$ rule, because δ's of matching "genesis" met each other. What happened is that the $\lambda_y \bowtie \delta_x$ step on ⋈ 2.22 created two δ_x, which started copying the Y subnet from two sides. At the last step of ⋈ 2.24, these two δ_x met and annihilated each other.

At that same step $\lambda_y \bowtie \delta_x$ of ⋈ 2.22, it would have been prudent to start introducing a fresh subscript z, so that as a final result we have a net with two differently-named subnets in order to avoid confusion. This is not critical for the λ and @ nodes since their interactions don't depend on the subscript. However, it is critical for the δ nodes, so as to be able to determine for a pair of δ's which of the two $\delta \bowtie \delta$ rules should be applied. Namely, at the $\delta_x \bowtie \delta_y$ step of ⋈ 2.24, we should have made the types of the two δ_y's different. But please note that in the same step we should keep the types of the two new δ_x's the same, so that they can annihilate each other in the final step. Determining whether to keep the types of two newly-generated δ's the same or to make them different is quite involved.

Therefore, our initial description of the translation of the λ-calculus glosses over one very important issue: determining the "scopes" of existence of related δ's, and when/whether to keep them related. The solution of this problem (including the dynamic assignment of δ types) can be implemented inside IN, which is a testament to their flexibility and their ability to capture all aspects of computation in a system. Lamping (1990) first studied this problem in the context of Optimal Lambda Reduction. His solution is very involved, and was later simplified and rationalized by Gonthier *et al.* (1992a) using ideas from Interaction Nets and the Geometry Of Interaction of Linear Logic. Recently, Guerrini (1997) studies the general theory of IN systems used for optimal

implementations of various calculi (*sharing graphs*). The use of INs as sharing graphs has been in our opinion the most successful application of INs to date.

2.5 Example: Infinite Streams in IN

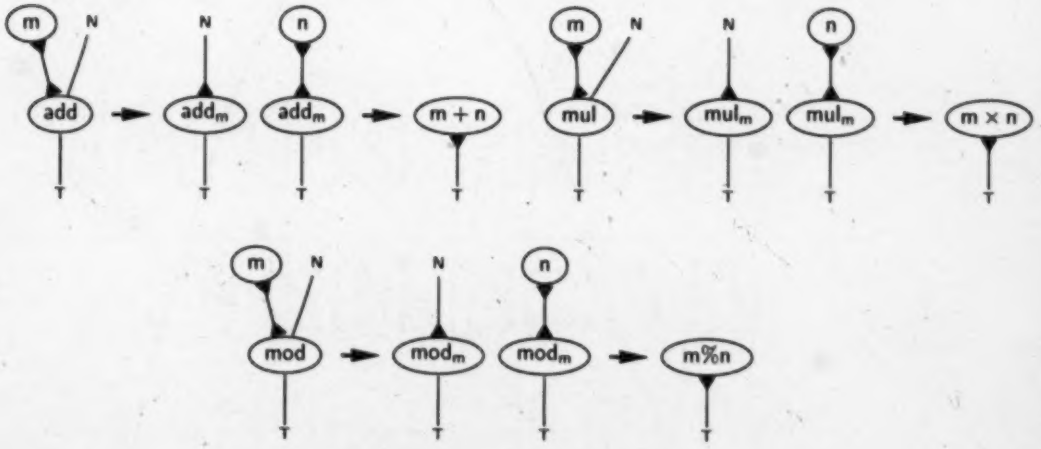
A topic of traditional interest in Data Flow architectures and in some programming languages is *potentially infinite data streams*. Some algorithms are most naturally expressed as if operating on infinite streams, and the ability of a language or framework to express infinite streams naturally is sometimes used as a measure of its expressiveness.

As an example of the power of IN, in this section we show how we can represent infinite streams over integer data.

2.5.1 Builtin Arithmetics

To simplify the exposition, we will implement numbers and elementary arithmetic operations with an infinite number of "builtin" interaction rules, instead of the unary arithmetic constructed in §2.2.1. The emphasis of this section is on how we can handle potentially infinite streams, so in order to keep us focused, we will assume our basic data elements to be built into the interaction system.

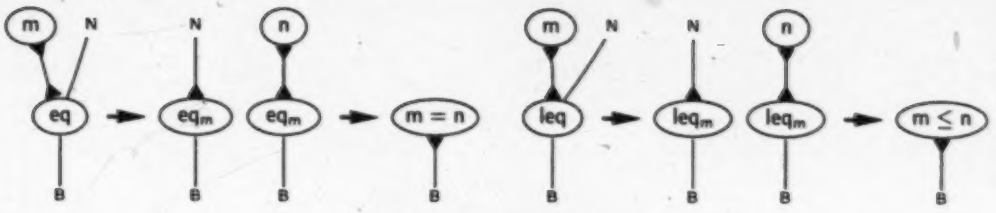
Our IN system will contain *numeric* nodes m for every integer m and *arithmetic* nodes add, mul and mod. It will also have *curried arithmetic* nodes add_m, mul_m and mod_m for every integer m, which are unary operators obtained from the corresponding binary operator by fixing one of the arguments to the constant m. These nodes are governed by the following enumerably infinite set of interaction rules. [⋈ 2.25]



⋈ 2.25: Built-in Arithmetic Operations.

Here m + n denotes the numeric node corresponding to the number $m + n$, and similarly for m × n and m%n (the latter being the modulo operation).

We also introduce *boolean* nodes eq, leq and *curried boolean* nodes eq_m, leq_m for every integer m. [⋈ 2.26]

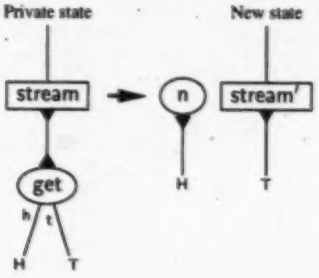


⋈ 2.26: Built-in Boolean Operations.

Here m = n denotes the boolean (true/false) node corresponding to whether the numeric relation $m = n$ holds or not, and analogously for m ≤ n.

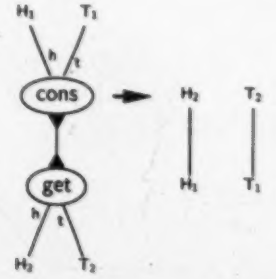
2.5.2 Infinite Numeric Streams

An infinite stream is a net (including a single node) that responds to get requests by returning some data item (number) n as the head of the stream, and another infinite stream as the tail. [⋈ 2.27]



⋈ 2.27: A Stream Responds to get Requests, Potentially *Ad Infinitum*.

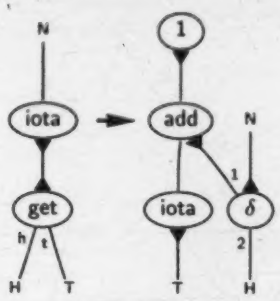
Sometimes we will use conventional cons to prepend an element to a stream. The following simple rule allows this: [⋈ 2.28]



⋈ 2.28: A Stream Constructor, cons.

One of the simplest possible streams is iota,[10] which returns as many numbers from the infinite sequence of natural numbers as is the number of get requests that it receives. [⋈ 2.29]

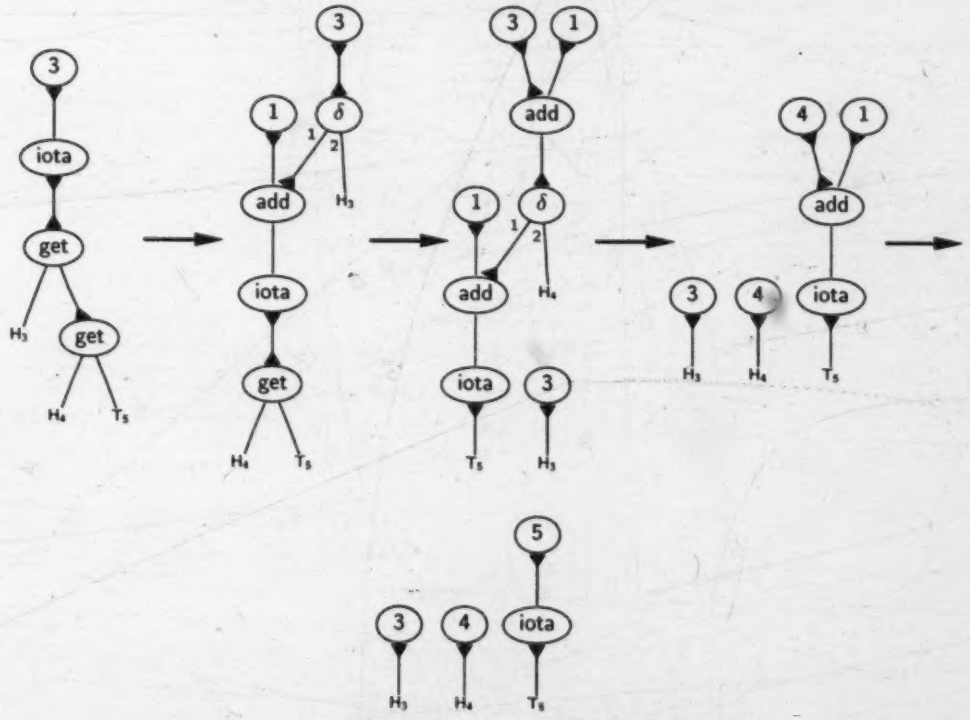
[10]The name iota comes from APL.



⋈ 2.29: The Stream iota Returns a Sequence of Natural Numbers Starting From N.

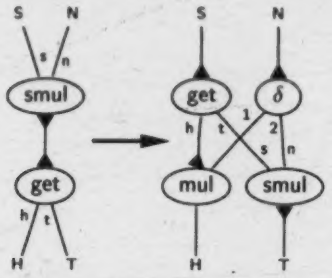
Below we denote iota having the number N at its auxiliary port as iota(N).

As an reduction example, here is how iota(3) returns (3, 4) and becomes iota(5): [⋈ 2.30]



⋈ 2.30: Obtaining Two Numbers from iota(3).

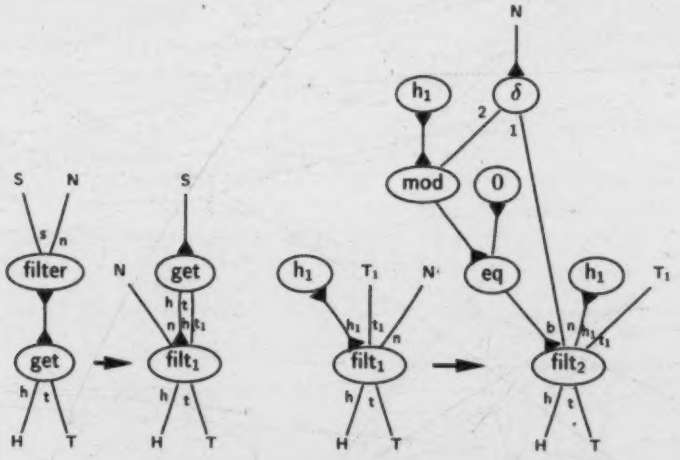
An example of a "stream transformer" is the *stream multiplier* smul which multiplies a given stream by a given number N: [⋈ 2.31]



⋈ 2.31: The Stream Multiplier smul.

The way smul works is simple: it gets the head of (an element from) the stream S, let's call this element h. It also duplicates the number N using the duplicator δ. Then h is multiplied by the first copy of N and the result is returned to H, as head of the resulting list. The tail T of the resulting list is formed as the stream multiplication smul of the tail t of S, multiplied by the second copy of N.

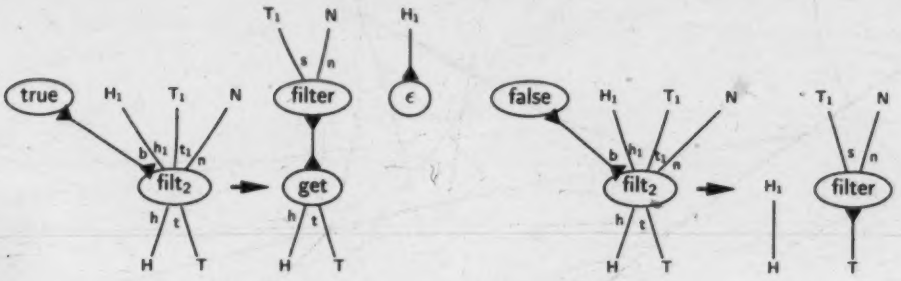
We also introduce a stream filter that removes all elements which are multiples of a given number N. When filter receives a get request, it steps through two transitional states filt_1 and filt_2. [⋈ 2.32]



⋈ 2.32: A Stream filter That Removes Multiples of N.

The first state filt_1 waits to get a number from the original stream S. Once the number h_1 is received, it is fed into a boolean test checking whether h_1 is a multiple of n.

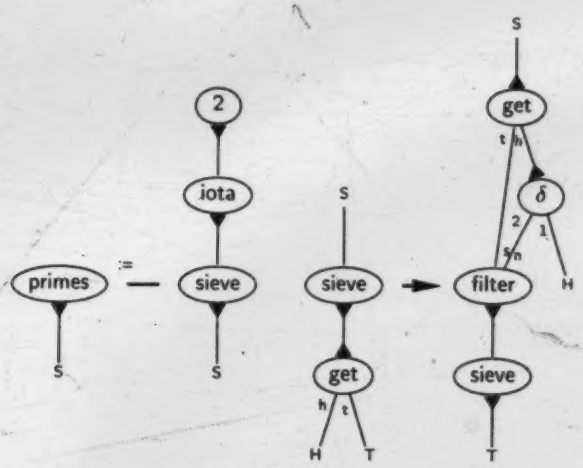
When filt_2 receives a boolean (true/false) on its principal port, it "makes a decision" based on the boolean. If it gets true (which means "skip this number"), it discards the number (H_1 in this case) and iterates to get and return the next number in the stream. If filt_2 gets false (which means "the number is ok"), it simply returns the number. [⋈ 2.33]



⋈ 2.33: filt_2 Skips or Returns the Number h_1.

2.5.3 Stream of All Prime Numbers

We are now ready to construct a stream primes of the prime numbers, using the Eratostene's Sieve algorithm: [⋈ 2.34]

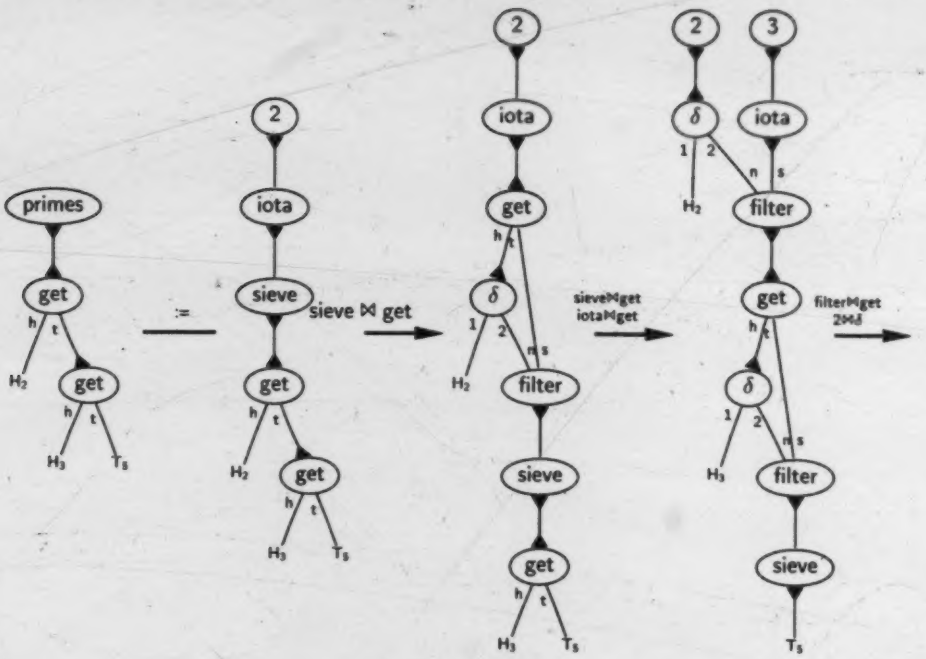


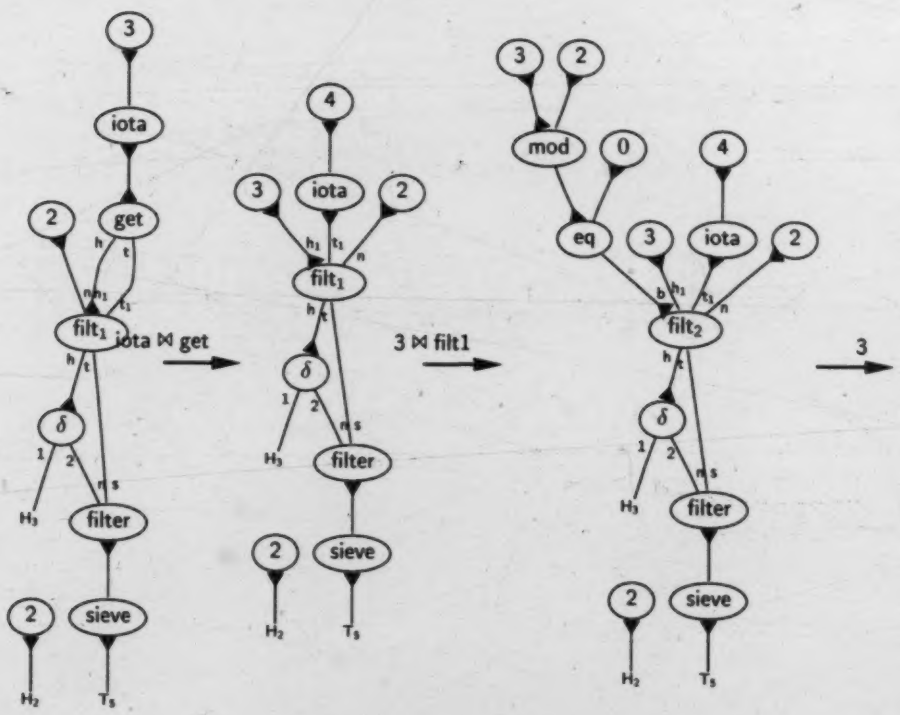
⋈ 2.34: A Stream Returning All primes Using *Eratostene's Sieve*.

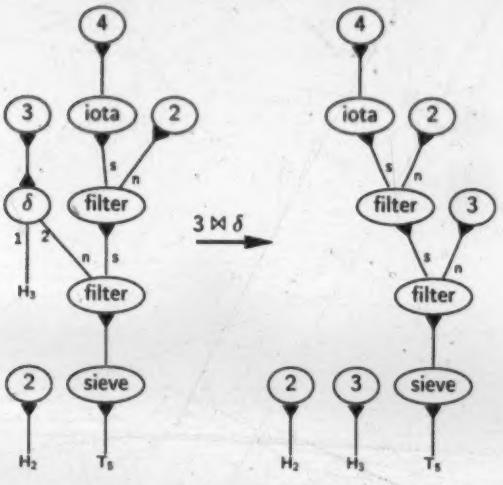
The stream primes is defined simply as an application of a stream sieve on iota(2). (We should not start from 1, because it would filter out all other numbers.) sieve works in this way: it gets and returns the first element of S as head of the result (H), but also duplicates this element, so that it can filter the tail T of the stream. It also tacks recursively another instance of sieve on T, so that it can sift through the rest of the stream.

A possible alternative is to put this instance of sieve between filter.s and get.t, instead at T. However, this will result in a less efficient implementation, since sieve will see many "garbage" (unfiltered) elements, and will produce a filter for every one of them. Let's denote by $p(n)$ the number of primes from 1 to n. To return all $p(n)$ primes, the implementation described above uses only $p(n)$ filters, while the alternative implementation would use n filters.

Below is an example of getting the first two elements from primes. [⋈ 2.35]


primes
:=
sieve ⋈ get
sieve⋈get
iota⋈get
filter⋈get
2⋈3


iota ⋈ get
3 ⋈ filt1
3

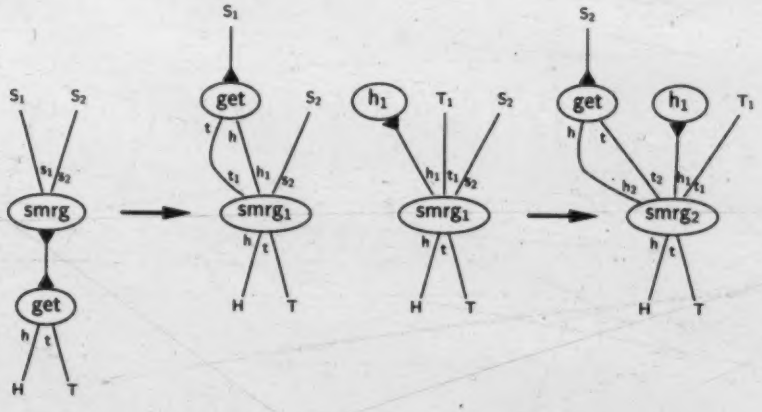


⋈ 2.35: Obtaining the First Two Prime Numbers.

2.5.4 Stream Merger

Now let's define a *stream merger* smrg which takes two sorted (monotonically increasing) streams on its ports s_1 and s_2 and merges them to another sorted stream that it "returns" to its principal port. Some of the numbers in the two streams might be the same, in which case the merger should discard the duplicates. For example, if s_1 is (2,4,6,8...) and s_2 is (3,6,9...), then the merged stream should be (2,3,4,6,8,9...) (the number 6 is returned only once).

The merger smrg works as follows. After it receives a get request, it proceeds in several intermediate steps. In the first step, smrg asks for an element from the first source stream s_1 and "becomes" $smrg_1$. [⋈ 2.36]



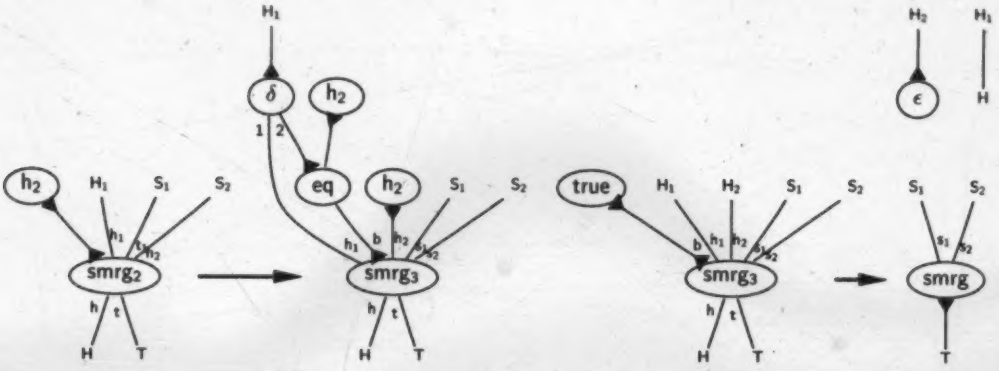
⋈ 2.36: Stream Merger smrg (Step 1).

In the second step, triggered by the received element h_1, $smrg_1$ asks for an element from the second source stream s_2 and "becomes" $smrg_2$.

In the third step, triggered by the second received element h_2, $smrg_2$ constructs a net that compares the two elements H_1[11] and h_2 for equality, and gives the boolean result of the comparison

[11] Since interaction rules are binary, we must use a variable H_1 instead of a node h_1, and we must use a δ to get two

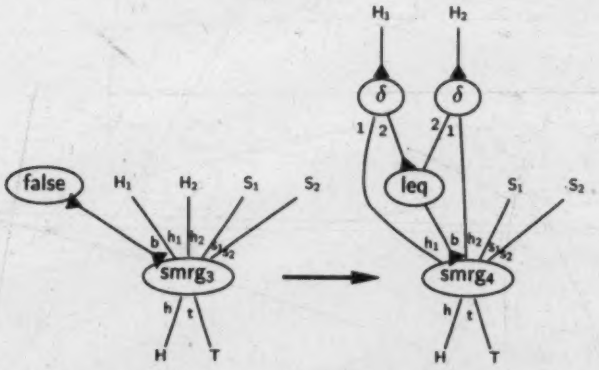
to smrg_3. [⋈ 2.37]



⋈ 2.37: Stream Merger smrg (Step 2).

After the boolean result of the comparison arrives, smrg_3 receives it on its principal port, and "makes a decision" how to proceed. If the two elements were equal, then smrg_3 returns one copy, discards the other, and goes back to the main smrg state.

If however the two elements were not equal, then smrg_3 must check which one is bigger than the other. It does this by constructing another comparison net, this time comparing for leq. [⋈ 2.38]

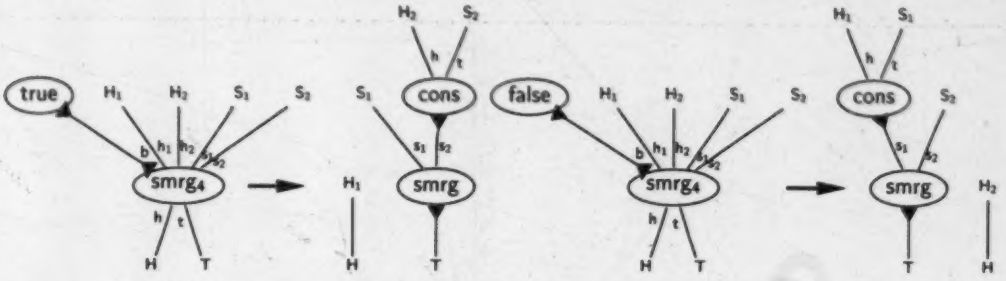


⋈ 2.38: Stream Merger smrg (Step 3).

Finally, after smrg_4 gets the boolean result of the comparison, it returns the smaller of the two elements to H. It "caches" the other element for further processing, by prepending it (using a cons) to the appropriate stream. [⋈ 2.39]

---

copies of H_1, one that is used in the comparison, and another that is passed to smrg_3. We could have used a similar δ for h_2, but as an optimization we use two copies of the node h2 on the RHS, since the RHS can contain any number of nodes.



⋈ 2.39: Stream Merger smrg (Step 4).

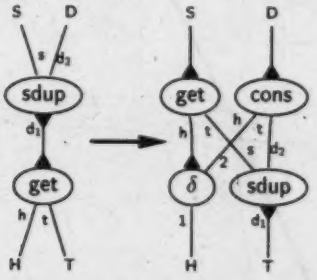
Note that smrg will use no more than one such auxiliary cons, because on every step it deconstructs the cached cons in order to re-examine the consed element.

2.5.5 Stream Duplicator

Similarly to the duplicator δ that we used to copy simple data items, we now introduce a *stream duplicator* sdup. It has a source port s and two destination ports d_1 and d_2, and copies the stream on its s port to both d_1 and d_2.

Unlike the simple duplicator δ which is eager (it copies whenever there is something on its s port), the stream duplicator sdup is by necessity lazy (it's impossible to actually copy an infinite stream). Therefore, sdup's principal port is not the source s, but one of the destination ports, say d_1.

sdup copies the stream one element at a time, initiated by get requests at d_1. The consumer at d_2 must wait for the consumer at d_1 to initiate duplication steps, and then d_2 can pick up the results which are presented as cons nodes. In other words, the consumer d_1 receives its copy immediately and as a result of its own get requests, while the consumer d_2 receives its copy in "reconstructed" form (using cons as constructor), and only as a secondary result of d_1's requests. [⋈ 2.40]

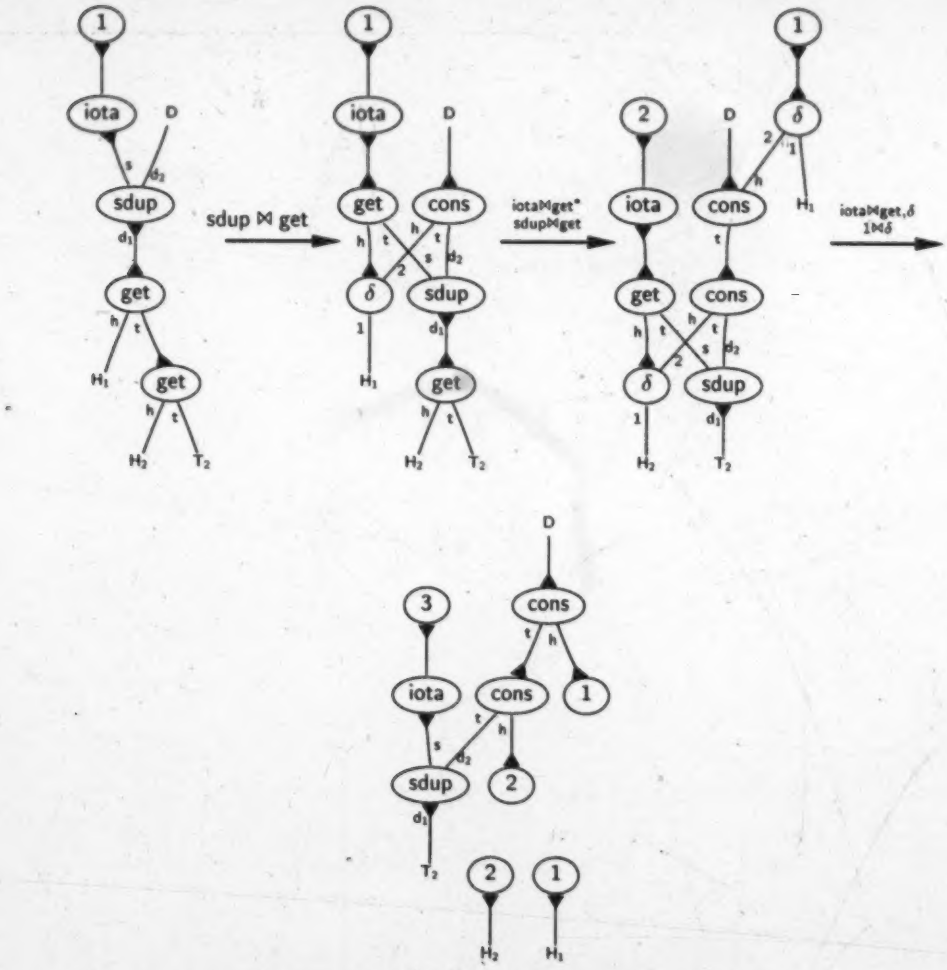


⋈ 2.40: The Stream Duplicator sdup.

Here D receives a cons of the actual head of the list (copied eagerly by δ), and the potential tail (copied lazily by sdup).

Note Above we arbitrarily assumed d_1 to be principal (active), and d_2 to be auxiliary (passive). Therefore, whenever we use sdup, we must prove that d_2 won't be "starved", *i.e.* left without a response, simply because d_1 doesn't need a response. This is perhaps the most subtle aspect of the example in §2.5.6 below.

Example: sdup ⋈ iota Let's see for example how does sdup duplicate a couple of elements from iota. [⋈ 2.41]



⋈ 2.41: Duplicating the Stream/Sequence iota.

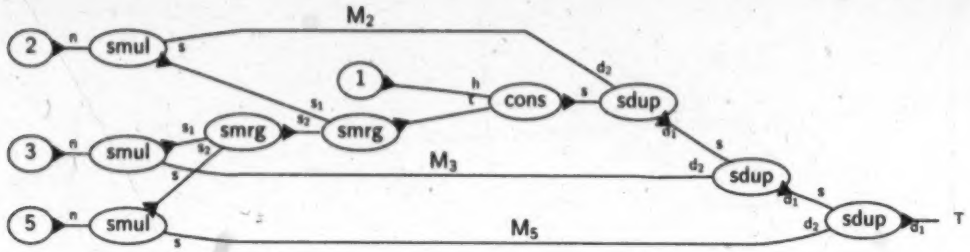
The two requested elements of the first stream (H_1, H_2, T_2) are returned immediately, while for the second copy D they are cached in a cons list, to be picked up by D at its leisure.

2.5.6 The Hamming Sequence

As a "grand finale" of this chapter, we give a more complicated example that involves circularity. It is a construction solving the following problem by Hamming:

> Generate all numbers of the form $2^i 3^j 5^k$ $(i, j, k \geq 0)$ in order and without duplication.

We use a standard idea: start from the number 1, replicate the output stream three times, multiply each duplicate stream by 2, 3 and 5 respectively, and merge the three multiplied streams to obtain the output stream. Stated explicitly, the output stream is fed-back into the net in order to generate more numbers. [⋈ 2.42]



⋈ 2.42: A Stream Solving the Hamming Problem.

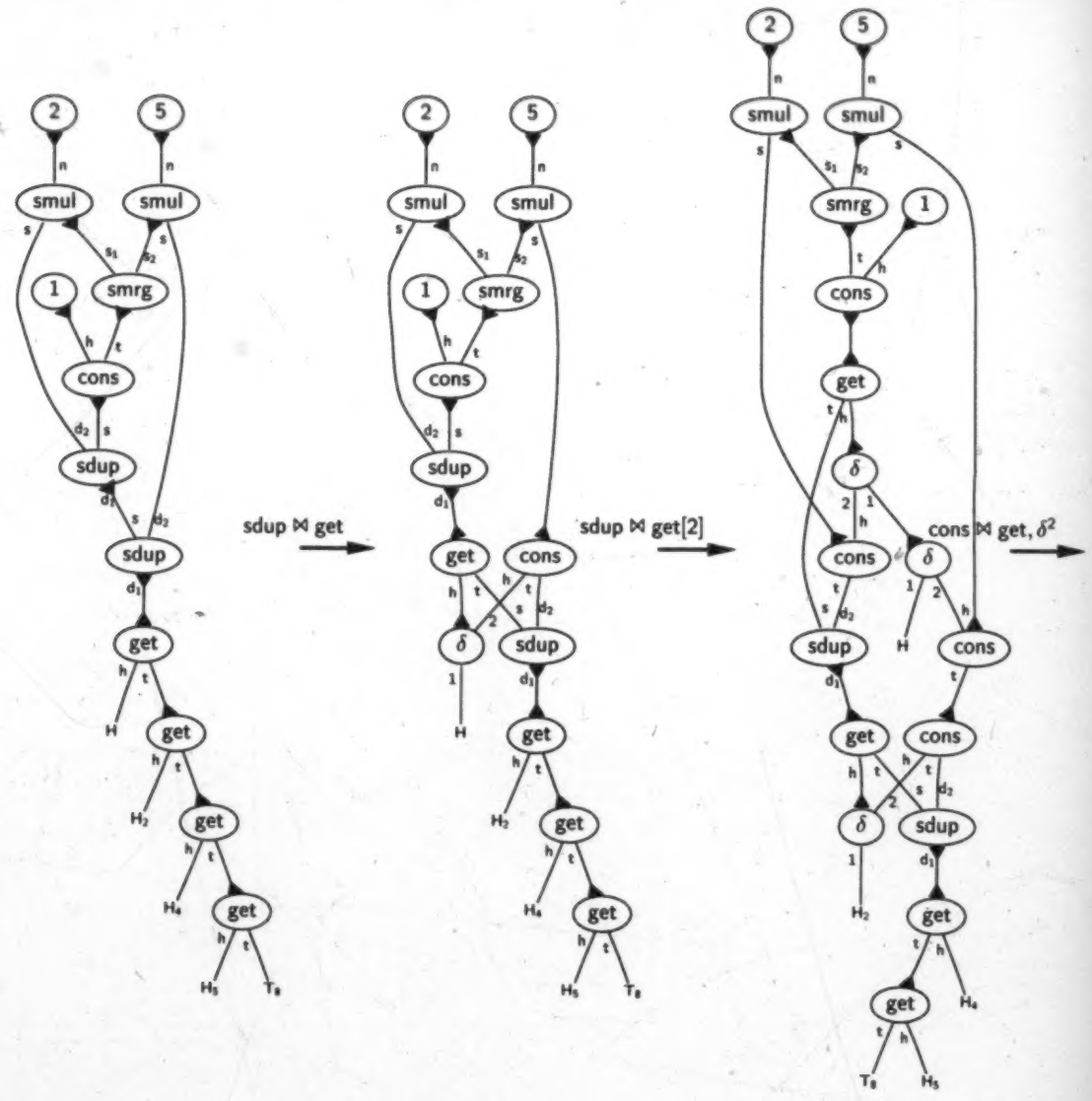
To understand how this works, it is imperative to remember that sdup takes its input (source) stream from s, not from its principal port d_1. What follows is a rough description of how this net works, not a proper proof that it does what it is constructed for.

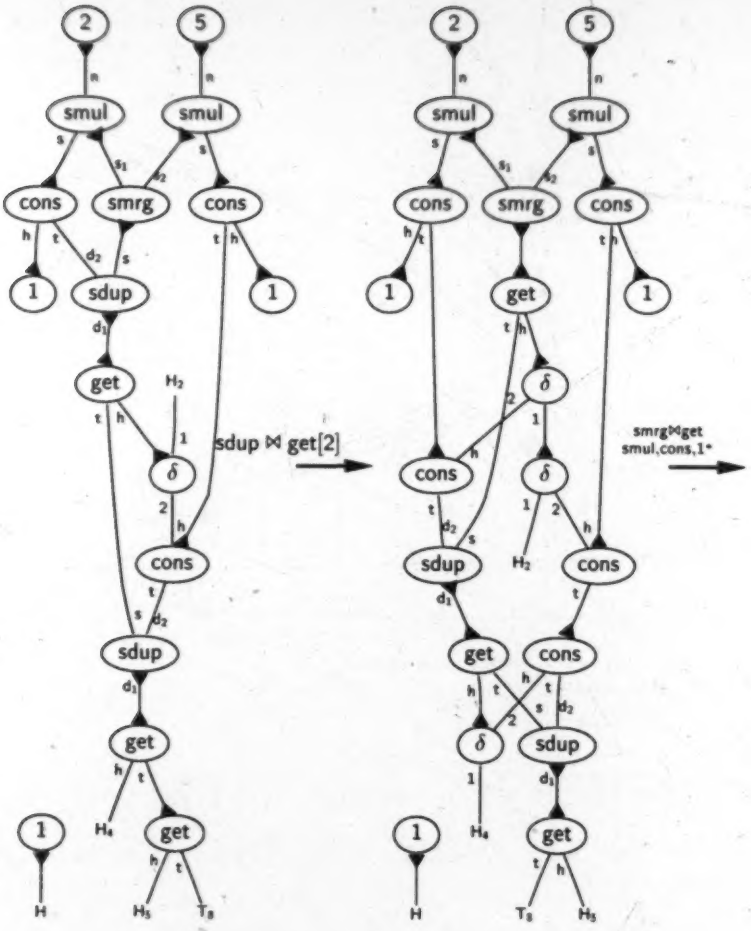
1. A get request is received at T.
2. The request "propagates" through the three sdups, and "leaves behind" simple (eager) δ's at the points M_2, M_3 and M_5.
3. The request reaches the cons, which returns the number 1 as the first result. The simple (eager) δ's duplicate this number 1 and place it at the s port of the three smuls as a first stream element. The dissolution of the cons also "shortcuts" the first smrg to the output.
4. The next get request activates the first smrg. The request then propagates through the second smrg, and so each of the three smuls is asked for an element.
5. The three smuls multiply the number 1 that was delivered to them in step 3 by the numbers 2, 3, and 5 respectively, and return the resulting numbers to the two mergers smrg.
6. The two mergers compare and "line up" the numbers in the correct order (2,3,5). However, only 2 is returned at this step.
7. Similarly to step 3, the returned number 2 is duplicated and delivered to the smuls (the points M_2, M_3 and M_5). In particular, the smul(2) produces the number 4.
8. ...
9. The number 4 "out-races" through (ordered) merging the number 5 that was produced earlier at step 5 and is returned to the output before the number 5.

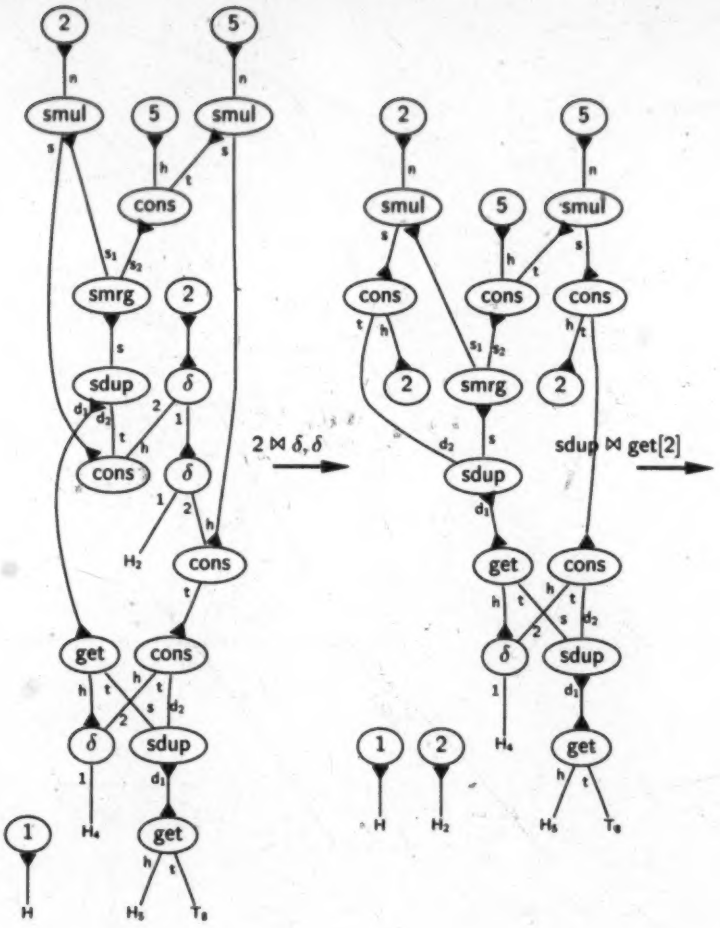
Although tedious, the arithmetical parts of the above construction are more or less obvious. The most subtle part is to show that there is no problem with the activation of sdups, namely that the internal (fed-back, looped) part of the construction never actively requires an element from the passive (d_2) port of the sdups, which would lead to dead-lock. We can prove that the system of Infinite Arithmetical Streams that we have introduced thus far is deadlock-free by constructing *switchings* for it, as in §2.3.3. We leave the proof to the reader.

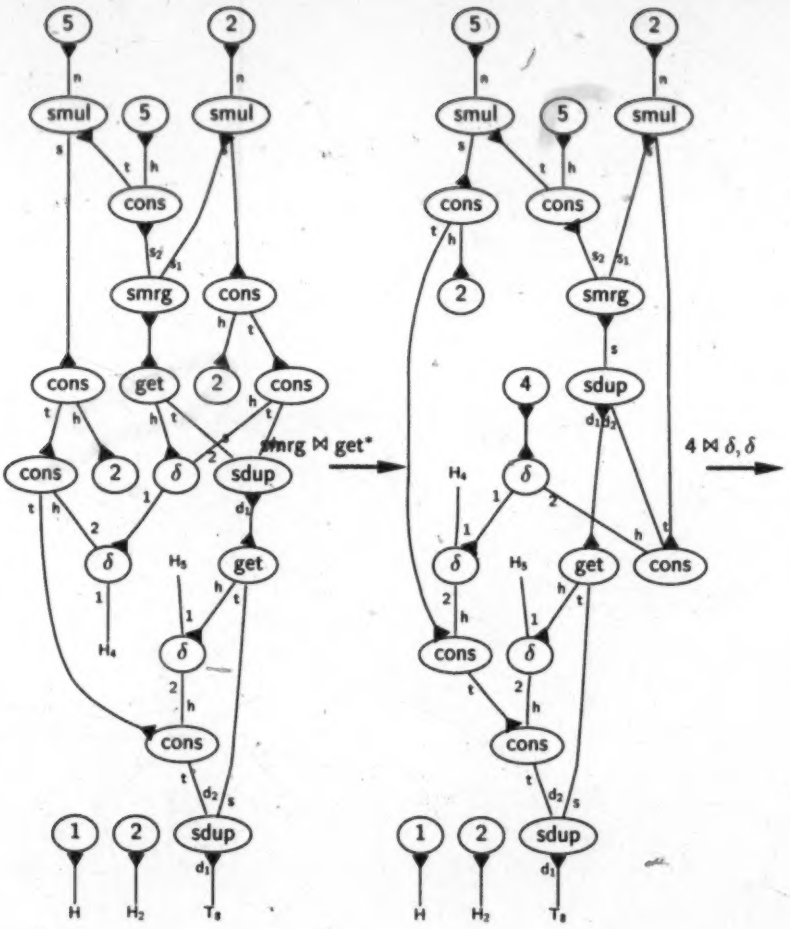
It is an interesting research problem to determine how does the number of nodes and steps-per-result grow with the length of the returned sequence. The trace below (⋈ 2.43) suggests that the growth is linear: the feedback part of the net stabilizes (doesn't grow), and the string multipliers smul get a simple copy of the output stream.

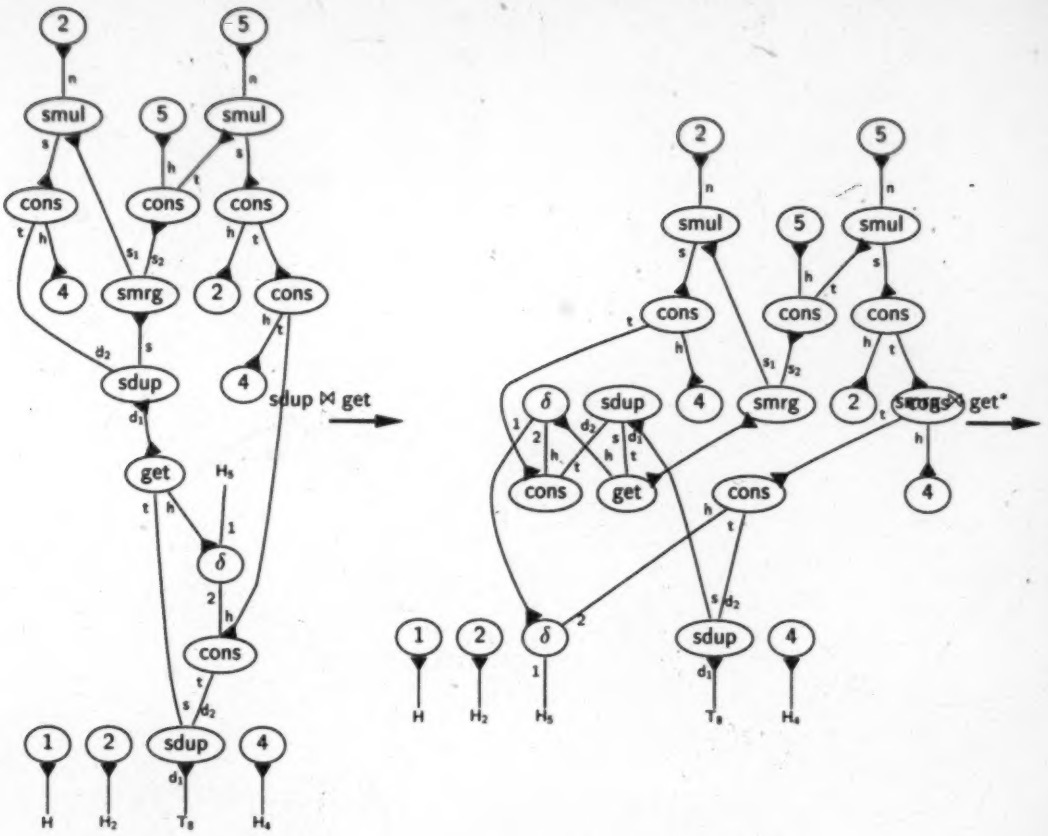
Without further ado (nor further explanation), we now show a trace of obtaining the first four numbers from a simplified version of the Hamming stream, corresponding to 2^i5^j [⋈ 2.43]

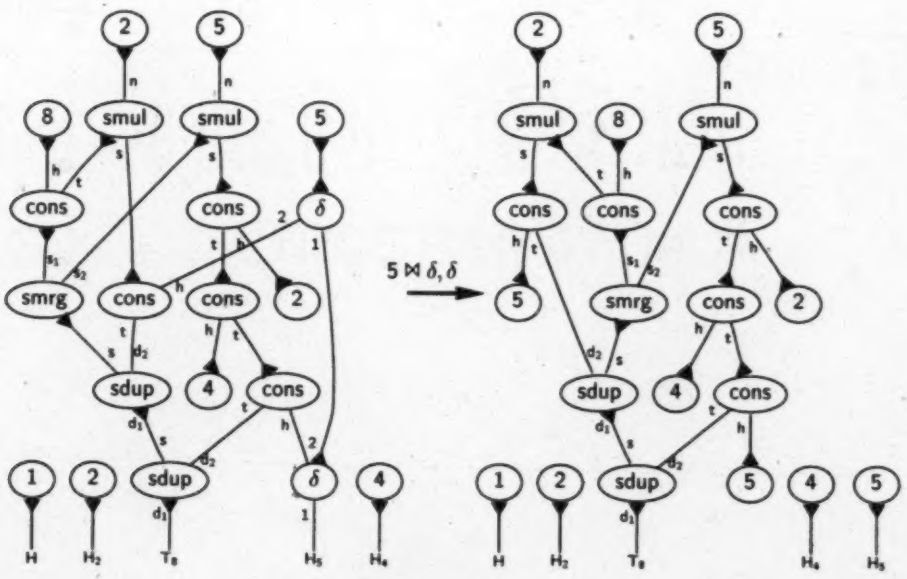

2
5
n
smul
s
s_1
s_2
1
smrg
h
t
cons
d_2
sdup
d_1
get
H
H_2
H_4
H_5
T_8
sdup ⋈ get
sdup ⋈ get[2]
cons ⋈ get, δ^2
δ
1
2


2
5
n
smul
s
s_1
s_2
cons
smrg
h
t
d_2
1
sdup
d_1
get
H_2
δ
sdup ⋈ get[2]
H_4
H
H_3
T_3
smrg⋈get
smul,cons,1*


2
5
n
smul
s
cons
h
t
s_1
s_2
smrg
sdup
d_1
d_2
δ
1
2
H₂
get
H
H₄
H₅
T₈
2 ⋈ δ, δ
sdup ⋈ get[2]


5
2
n
smul
5
smul
s
t
h
cons
s_2
s_1
smrg
cons
h
t
cons
get
2
cons
t
h
cons
2
δ
sdup
1
2
d_1
δ
H_5
get
1
h
t
H_4
δ
2
h
cons
t
d_2
s
1
2
sdup
d_1
H
H_2
T_s
smrg ⋈ get*
4
H_4
1
4 ⋈ δ, δ


sdup ⋈ get
get*


5 ⋈ δ, δ

⋈ 2.43: Obtaining the First Four Elements From a Simplified Hamming Stream.

The next two numbers to be returned from the stream are:

- 8, which is "cached" above the smrg.
- 10, which will be returned by both smuls since $10 = 2 \times 5 = 5 \times 2$, and then collapsed to one copy by smrg.

Chapter 3

Non-Determinism in Interaction Nets

The strong confluence of conventional Interaction Nets accounts to a large extent for the simplicity and attractiveness of the formalism, and for some of its deeper theoretical aspects, such as deadlock-freeness of certain syntactically-identifiable fragments. However, it also means that IN cannot be used to model and implement *non-deterministic* systems such as concurrent object-oriented systems or the π-calculus. In such systems an entity (object/agent/process) typically can be connected to more than one other entity, and the interaction with one of them may well preclude interaction with others. For example, consider the π-calculus process $c!a, c!b, c?x.P$.[1] It can reduce to either $c!b, P[a/x]$ or $c!a, P[b/x]$, which are not equivalent if a and b are different names.

One can consider various ways of extending INs with non-determinism, by "breaking" one or more of the definitions in §2.1. We prefer models that differ minimally from conventional INs, in hopes that it will be possible to salvage some of the nice theoretical results about IN in the new enriched setting. In this thesis we consider the following ways of introducing non-determinism.

IN with Multiple (reduction) Rules (INMR) Allow more than one reduction rule per redex. The choice of which reduction rule fires is non-deterministic.

IN with Multiple Principal Ports (INMPP) Allow more than one principal port per node. If there is more than one principal ports connected to the principal ports of a node, the choice of which pair interacts is non-deterministic.

IN with MultiPorts (INMP) Allow more than one connection per port. If more than one principal port is connected to a given principal multiport, the choice of which redex interacts is non-deterministic.

IN with Multiple Connections (INMC) Allow hyper-edges (in the graph-theoretical sense), *i.e.* connections between more than two ports. If more than one principal ports are involved in such a multi-connection, then the choice of which pair interacts is non-deterministic.

It is a natural question how the expressive powers of these methods of introducing non-determinism in IN compare to each other, and on the other hand how far does each one depart from conventional IN. In this chapter we define formally the methods enumerated above, and in the next chapter give some informal results about inter-representation of various methods.

We leave to future work the investigation of the theory of non-deterministic INs, as well as the question which conventional IN results can be transferred to this enriched setting.

[1] See §5.1 for an introduction to the the π-calculus, and the notation that we use: $c!a$ is output, and $c?x$ is input

Are these four all the possible reasonable extensions of IN for the purpose of introducing non-determinism? We cannot prove this, but we think that we have a good understanding of the "choice space". First of all, we can introduce non-determinism "at the source", *i.e.* in rule application:

- Relax the restriction "one rule per redex, applicable in only one way". This leads to INMR.
- Allow more than two nodes in a redex. This leads towards general graph rewriting.

From a different viewpoint, IN are a (restricted) class of graphs, and one can modify various aspects of graph elements:

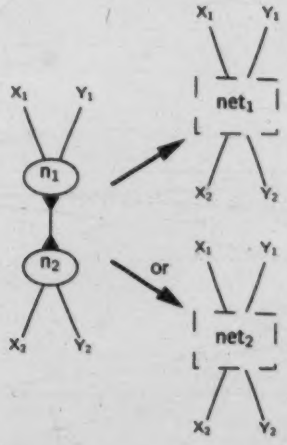
- Looking at the locality of a node:
  - It does not seem very useful to remove the notion of *ports* and allow edges to attach "anywhere" to a node. (However, later in our investigations we naturally come to a notion of such kind of node, the *connector* node (see *e.g.* §4.7). We can model such a node in INMP as a node with a single principal multiport.
  - We can relax the restriction of linearity, *i.e.* only one connection per port. This leads to INMP.
- Looking at the type of edges, we can allow hyper-edges. This leads to INMC.

Of course, generalizations based on wholly different classes of ideas will likely come to light. For example, Fernández and Mackie (1996a) introduce some new entities (*memory* and pseudo-edges), and let conventional IN manage the access to memory cells by moving and manipulating pseudo-edges. However, it is our opinion that this strays unnecessarily far from IN.

3.1 IN with Multiple (reduction) Rules (INMR)

Conventional INs do not allow more than one interaction rule per redex (cut), and in the case when the cut is symmetric (*i.e.* when the two nodes in the cut are of the same type), the RHS of the rule has to be symmetric as well. This means that a redex can be reduced in only one way.

We introduce INs with Multiple (reduction) Rules (INMR), whereas we allow more than one applicable reduction rule per redex. The choice of which reduction rule will fire is non-deterministic. [⋈ 3.1]



⋈ 3.1: IN with Multiple (reduction) Rules (INMR).

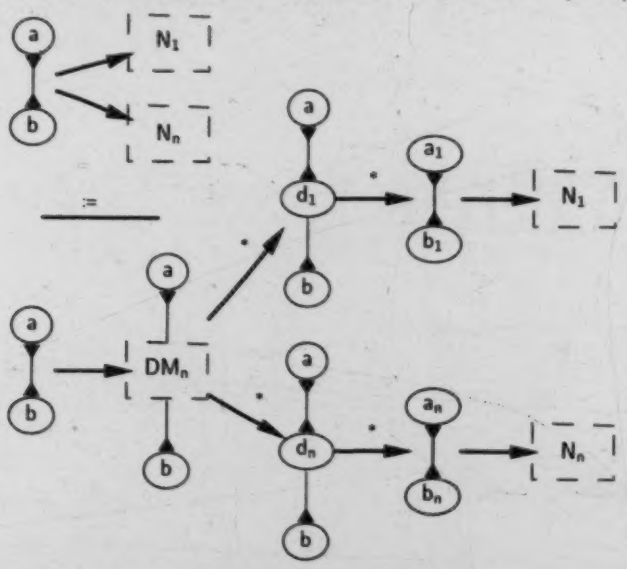
Sometimes we may use priorities between rules over the same redex. In the case when no priorities are specified, we assume that rule selection is fair, *i.e.* an applicable rule has a non-zero probability of being selected.

3.1.1 Using Asymmetric Reflexive Rules to Capture INMR

In this section we develop a translation from INMR to a restricted system that only allows one particular type of INMR rules, asymmetric reflexive rules. We call this target system Interaction Nets with Asymmetric Reflexive rules (INAR). INAR does not allow different outcomes for the interaction of two nodes. The only variation from conventional IN that INAR allows is that a reflexive rule (where the LHS is symmetric, such as $\mathsf{a} \bowtie \mathsf{a}$) may have an asymmetric RHS. The interest of this translation is that the seemingly poorer system INAR can capture[2] the richer system INMR faithfully

Take an INMR system S, and consider a redex $\mathsf{a} \bowtie \mathsf{b}$ of S that has n applicable rules $\mathsf{a} \bowtie \mathsf{b} \rightarrow N_1, \ldots, \mathsf{a} \bowtie \mathsf{b} \rightarrow N_n$. Let N be the maximum of all numbers n over all redexes of S. The target INAR system T will have the node types of S and extra node types *decision maker* dm, N types of *decision* nodes $\mathsf{d}_1, \ldots, \mathsf{d}_N$, and some variants of these nodes subscripted with numbers $1 \ldots N$.. The translation of S nets is very simple, it is the identity function. The essence of the INMR→INAR translation is in the translation of rules.

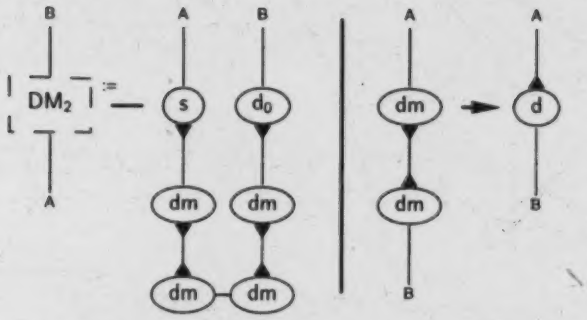
The idea is to take the "decision" of which rule will be applied, away from the redex $\mathsf{a} \bowtie \mathsf{b}$, and give it to an explicit "decision making" net DM_n. This net should be able to produce any one of n possible decisions d_i, from which the n possible outcomes N_i will follow: [⋈ 3.2]



⋈ 3.2: Translating INMR to INAR: Basic Idea.

The construction of a decision maker net DM_n that can produce any of the decisions d_i is an easy exercise from the theory of 1-dimensional INs. (Please see (Lafont, 1991) for a thorough investigation of this topic, called there Interaction Strings.) One possible construction is to make DM_n a chain of n pairs of decision making nodes facing each other, and terminated on one side with a decision node d_0, and on the other side with a *sink* node s. The INAR rule on the right side of the following figure states that there are two possible outcomes from a pair $\mathsf{dm} \bowtie \mathsf{dm}$: one is a decision node d with its principal port turned towards s, and the other is a d turned towards d_0. [⋈ 3.3]

[2] See §4.1 for a discussion of implementation translations between systems.



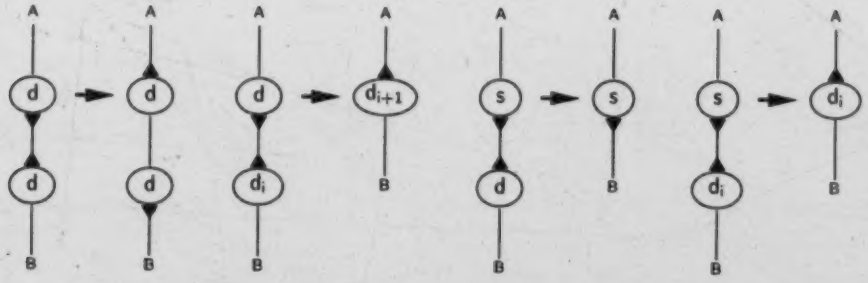
⋈ 3.3: Translating INMR to INAR: Decision Making.

Note that since DM_n is asymmetric, in order to make our translation deterministic, we need to order the nodes of the source system S, and to assume on the schema ⋈ 3.2 that (for example) a is before b in that order.

After DM_n "settles" (all decisions are made), we only have to count the number of d nodes looking towards the d_0 node. This number will give us the final decision d_i. The rules below perform the following tasks:

1. Decision nodes facing each other d ⋈ d "jump over" each other, so that they percolate towards the corresponding terminator.
2. The sink terminator s simply kills d nodes, because we do not care to count d nodes looking towards it.
3. The decision terminator d_0 counts the number of d nodes looking towards it, by increasing its subscript.
4. Finally, the decision d_i removes the sink s.

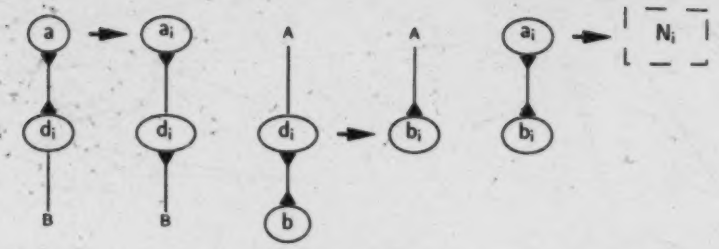
[⋈ 3.4]



⋈ 3.4: Translating INMR to INAR: Counting d Nodes Looking Towards d_i.

It is easy to see that these rules together with the rule ⋈ 3.3 will reduce the decision making net DM_n to a single decision node d_i, where $i \in \{1 \ldots n\}$.

After the decision d_i is produced, it is easy to make a and b use it to produce the final result N_i: [⋈ 3.5]

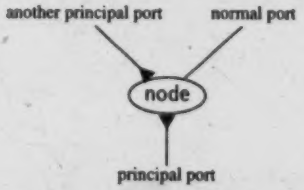


⋈ 3.5: Translating INMR to INAR: Using the Decision d_i.

(We also need to replicate the first two rules for b and a respectively, because in a different situation they may have to play each other's role.)

3.2 IN with Multiple Principal Ports (INMPP)

In conventional INs every node has only one principal port through which it can interact. We consider INs with Multiple Principal Ports (INMPP), for which we allow more than one principal port per node. Bawden (1992) studies *Connection Graphs* which are similar. [⋈ 3.6]



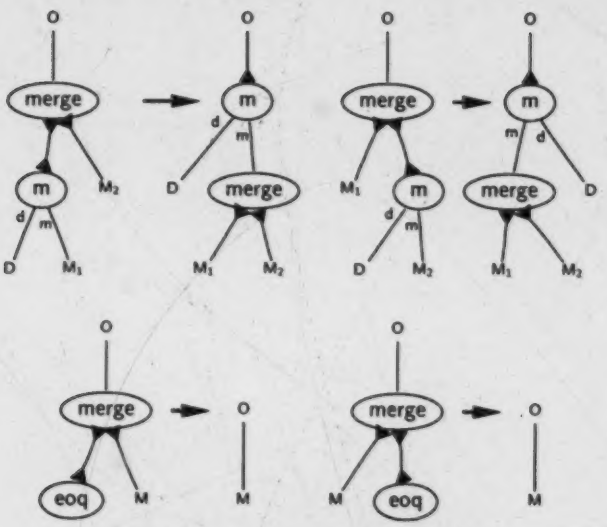
⋈ 3.6: IN with Multiple Principal Ports (INMPP).

If there are principal ports connected to two principal ports of a given node n, then the choice of which pair interacts is non-deterministic. After interaction, n may change type in the result and/or the non-selected principal port may become non-principal, which may prevent the non-selected interaction from ever occurring.

3.2.1 INMPP Example: Queue Merger

The paradigmatic INMPP example is a *queue merger* merge. This node would be useful in an IN system modeling Concurrent Object-Oriented Programming. Objects accept various kinds of *messages*; for this example we assume there is only one kind m. A message has its principal port towards the destination object, and it has two auxiliary ports: d which carries the data of the message, and m which holds further messages towards the same object, sequenced in a message queue.

The purpose of merge is to merge (interleave) two independent message queues towards an object. Since it has to be attentive towards both message queues, merge has two principal ports. [⋈ 3.7]

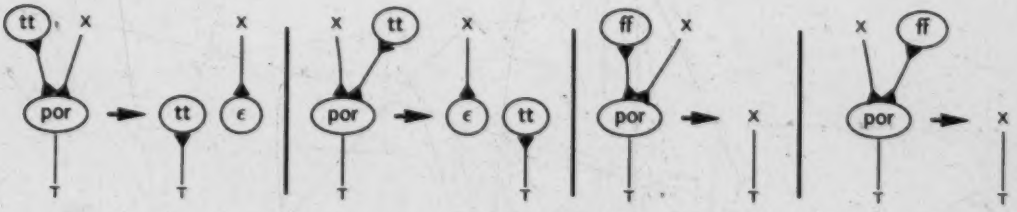


⋈ 3.7: INMPP Example: Merger.

Optionally, we have also introduced above a special message **eoq** (end-of-queue) that closes a message queue and removes (short-circuits) a merger.

3.2.2 INMPP Example: Parallel Or

Below is an example of an INMPP. It is a short-circuiting (non-strict) logic element or, also known as *parallel or*. [⋈ 3.8]



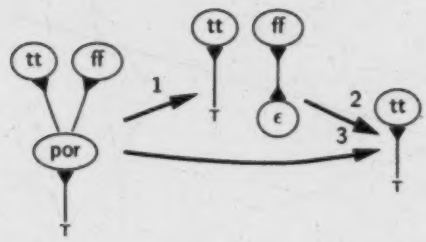
⋈ 3.8: Parallel Or as an INMPP.

Here the principal ports of por carry its arguments and the other port carries the result. tt and ff are the logic constants, and the eraser ϵ was introduced in §2.2.

The example works as follows:

- If one of the arguments is tt then the result of the por is set to tt, and the other argument is consumed using an eraser (the first two rules). Although the interaction will not be "completed" until that second argument is seen and erased, the result port is determined and can be used as soon as the determinant tt argument is seen.
- If one of the arguments is ff then the result port of por is short-circuited to the other argument (the last two rules).

It is interesting to see what happens when two rules are active simultaneously, *e.g.* when por has both tt and ff as arguments. [⋈ 3.9]



⋈ 3.9: Parallel Or Connected to Both tt and ff.

In this case the result does not depend on the order of rule selection, *i.e.* we have used the multiple principal ports only to introduce extra parallelism in the evaluation process, not for "real" non-determinism. Below we will use the non-deterministic properties of INMPP substantially.

3.3 IN with MultiPorts (INMP)

In conventional INs every port has exactly one link (edge) connected to it. In order to model in the most natural way systems where several "clients" can be connected to the same "server", we introduce IN with MultiPorts (INMP), these being ports that can accommodate more than one connection. If more than one principal port is connected to a given principal multiport, the choice of which redex interacts is non-deterministic.

We denote multiports with a bold dot at the port. The property of being a multiport is independent of the property of being a principal port, so a port may have one of four types:

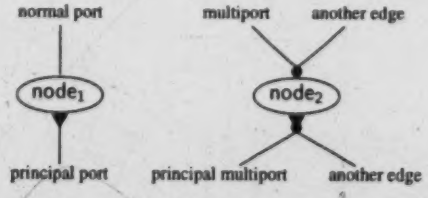
Normal port (no symbol).

Multiport (dot).

Principal port (triangle).

Principal multiport (triangle-dot).

Most often when we say "multi" or "principal", we do not exclude the other property, so we will say things like "non-principal multiport" when we want to be exact. [⋈ 3.10]



⋈ 3.10: IN with MultiPorts (INMP).

3.3.1 Uniform Treatment of Links

An INMP rule can only specify the evolution (re-wiring) of a finite number of edges, but if the LHS contains multiports, then the total number of edges on the LHS is not bound a-priori. Therefore, we require INMP rules to treat all but a finite number of links uniformly, as defined by the following principles:[3]

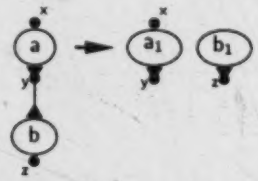
[3]Below, when we say "multiport", we mean either a principal or an auxiliary multiport.

1. Every multiport occurrence[4] x on the LHS (source) should have a corresponding multiport occurrence x′ on the RHS (target). All potential edges of the source x that are not explicitly given in the LHS are migrated to the target x′ *en-masse* during rule application.

2. The applicability of a rule does not depend on the current arities of multiports, unless a constraint is explicitly noted (see §3.3.2). Such constraints may only make a finite number of distinctions.

3. Implicit merging of multiports is not allowed, *i.e.* two sources x_1 and x_2 on the LHS can't map to the same target x′ on the RHS. If we need to express port merging, we have to migrate the edges one by one, using iterative applications of appropriate rules.

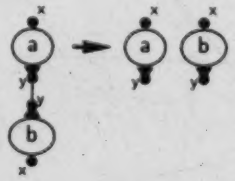
The correspondence between port occurrences x and x′ is usually established by name (the names of the two ports are the same). In the sample rule below, all potential edges connected to a.x are migrated to a_1.x, those of a.y to a_1.y, and those of b.z to b_1.z. [⋈ 3.11]



⋈ 3.11: Edges are Migrated Between Ports of the Same Name.

Note that the edge a.y ⋈ b (the cut) is consumed by the rule, but the multiport a.y is not, it is "transferred" to the RHS multiport a_1.y. Indeed, since we cannot ensure that a.y will not hold more than just the cut edge, we cannot afford to obliterate it on the RHS, or else the extra edges will become dangling. For example a rule with the above LHS and a RHS of x−z is not permissible.

If there are same-named multiports on the LHS, then the ambiguity is resolved by migrating edges between nodes of the same name (type). In the sample rule below, all potential edges of a.x from the LHS are migrated to a.x on the RHS (not to b.x), and similarly for a.y, b.x and b.y. (The cut edge a.y ⋈ b.y is consumed and is not carried over to the RHS.) [⋈ 3.12]



⋈ 3.12: Edges are Migrated Between Nodes of the Same Type.

We also try to position corresponding ports in visual parallel.

3.3.2 Arity Constraints

In the previous section §3.3.1 we described how INMP rules act on the "bundle" of edges connected to a multiport. Namely, all such edges (with the possible exception of the cut edge) are migrated

[4] That is not subject to **zero** or **one** constraints, see §3.3.2.

from a port on the LHS to a port on the RHS. This merely restates the conventional IN principle that the two redex nodes are removed from the net and are replaced with the net on the RHS, while their free ports (the ones not participating in the cut) are mapped to ports of the replacement net.

We often will need to merge two "edge bundles" incident with two different multiports in a net. We cannot do this through the application of *one* INMP rule (see clause 3 of §3.3.1). Therefore, we will accomplish such merging by applying a rule iteratively. But in order to avoid infinite looping, and to allow the application of a "finalization" ("clean-up") rule, we need to:

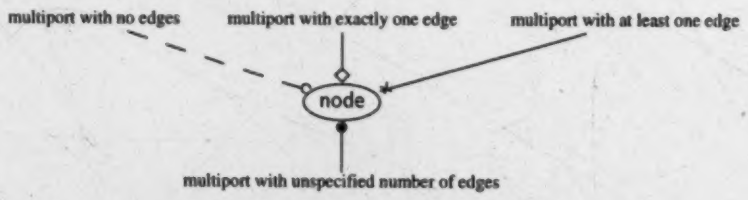
- Qualify rule applicability with the current arity of (number of connections incident with) a multiport. We call such additional restrictions *arity constraints*.
- Allow the specification of more than one rule for a given redex.

We use only a few types of arity constraints:

none	no constraint,
zero	zero incident edges,
one	exactly one incident edge, and
positive	at least one incident edge.

Table 3.1: INMP Arity Constraints

We introduce the following notation: [⋈ 3.13]

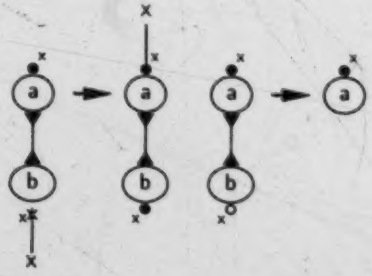

multiport with no edges
multiport with exactly one edge
multiport with at least one edge
node
multiport with unspecified number of edges

⋈ 3.13: INMP Arity Constraints.

Please note that the arity constraint symbols do not denote extra port types, rather they denote particular port "states".

We don't bother to denote with + the principal multiports incident to the cut edge of a rule, since we already know that they have at least the cut edge. Neither do we bother to denote with + multiports on the RHS for which we can deduce that they will have at least one edge after the application of a rule.

As an example of the application of arity constraints, the iterative application of the left rule below migrates the b.x edges to a.x, after which the right rule makes b disappear: [⋈ 3.14]



⋈ 3.14: Migration of the b.x Edge Bundle to a.x.

In the left rule above,the port b.x on the LHS has constraint **positive**, so the rule cannot be applied unless there is at least one edge connected to it. On the RHS that edge is transferred to a.x, and furthermore we have shown b.x with a constraint **none**. The reason is that without further information, we cannot deduce any stronger constraint about its state.

Important Note In order not to clutter our diagrams unnecessarily, we often omit multiports if there are no edges connected to them. We denote a multiport with an empty dot only when we want to emphasize that the **none** constraint applies to it.

3.3.3 Rule Priorities

The arity constraints of the example above partition the "state space", in the sense that they are mutually-exclusive, and that they cover all possibilities. The port b.x can either have **zero** or a **positive** number of incident edges.

Note Rule application depends only on constraints in the LHS. On ⋈ 3.14 we have "volunteered" some information about the possible state of the RHS, but these constraints on the RHS do not play a role in the reduction process.

However, we won't be able to always limit ourselves to the use of such "ideally partitioned" rule sets. For example, the constraints **one** and **positive** are not mutually-exclusive. Therefore, two rules that utilize these constraints will be simultaneously applicable in some cases. We still want to impose some restrictions on the simultaneous applicability of multiple rules, because we don't want to re-enter the realm of INMR (§3.1). For this purpose, we introduce *rule priorities*.

We will use the most natural kind of prioritization, where the priority is determined by *specificity*. We denote that constraint c_1 is less specific than constraint c_2 with the precedence symbol: $c_1 \prec c_2$. We define the specificity ordering of the constraints as

$$\textbf{none} \prec \textbf{zero} \qquad \textbf{none} \prec \textbf{positive} \prec \textbf{one}.$$

Then the priority of a rule is determined by the priorities of the set of constraints that it imposes on multiports of a redex. We will be careful to use only rule sets that are totally-ordered by such priority, for every possible redex instance. We will always choose for execution the highest-priority rule that satisfies the constraints.

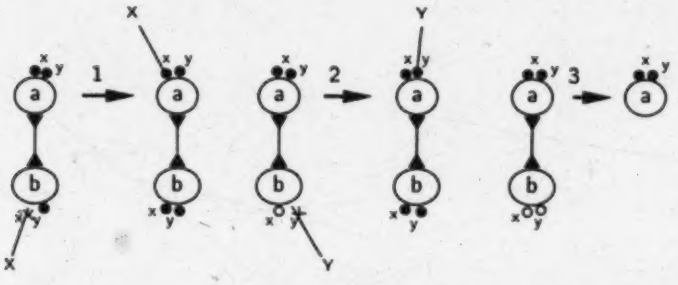
More formally, let the set of multiports of a redex ρ be P. Let the set of rules with the same LHS be R, and the subset of those rules that are applicable to ρ be $R_\rho \subset R$ (ρ may not satisfy the constraints imposed by some of the rules). Then for every rule $r \in R_\rho$, consider the set of constraints $C(r) \stackrel{\text{def}}{=} \{C(r,p) \mid p \in P\}$ that it imposes on each of the multiports p. The priority relation between two constraint sets $C(r_1)$ and $C(r_2)$ is defined thus:

$$C(r_1) \prec C(r_2) \quad \text{iff} \quad \forall p \in P.\, C(r_1,p) \prec C(r_2,p).$$

We will use only rule sets R such that for *every* redex instance ρ, the set of constraints $C_\rho = \{C(r) \mid r \in R_\rho\}$ is totally-ordered. (However, we will not prove this formally in every case, often we will leave it as an exercise to the reader.)

For example, consider again the link migration of ⋈ 3.14. Since the two rules are mutually-exclusive, the sets R_ρ are either empty or singletons, so they are trivially totally-ordered.

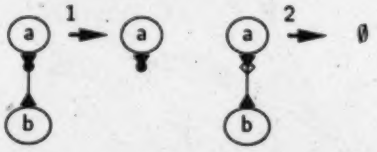
Now consider the task of migrating two edge bundles x and y from node b to node a. We could do it in this way: [⋈ 3.15]



⋈ 3.15: Migration of Two Edge Bundles Directed by Rule Priorities.

Here again, the three rules are mutually-exclusive, so the condition on priorities is satisfied trivially (we don't need priorities). (Note: in this example, wi did not bother to calculate any constraints about the state of ports in the RHS. These are trivial to calculate, for example in the middle rule b.x continues to hold the **zero** constraint, while a.y transitions from **none** to **positive**.)

However, another common task that we encounter in our applications uses priorities substantially. That is the task of exhausting a primary edge bundle (the edge bundle attached to the principal multiport of node a), and then progressing to a new state a′ (or disappearing). For this task, we need to treat specially the last edge in the bundle, thus we need the **one** constraint. This constraint is not mutually-exclusive with the **positive** constraint (which is always implied for the cut edge).. [⋈ 3.16]



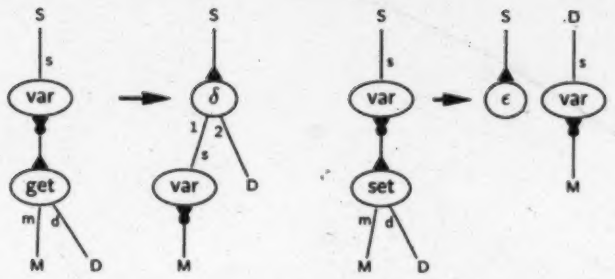
⋈ 3.16: a Exhausts the Primary Edge Bundle, then Disappears.

Here the second rule r_2 is applicable always when the first rule r_1 is, and also r_2 is always more specific than r_1 ($C_\rho(r_1) \prec C_\rho(r_2)$ for every ρ). The reason is that r_1 implies the **positive** constraint (the cut edge makes "at least one" edge), and **positive** $\prec$ **one**. If we didn't have priorities, at the moment when a had only one edge, rule r_1 might have fired, leaving as garbage a node a without any edges.

3.3.4 Example: Variable (Reference)

We can easily represent typical object-oriented notions such as *state change* in INMP.

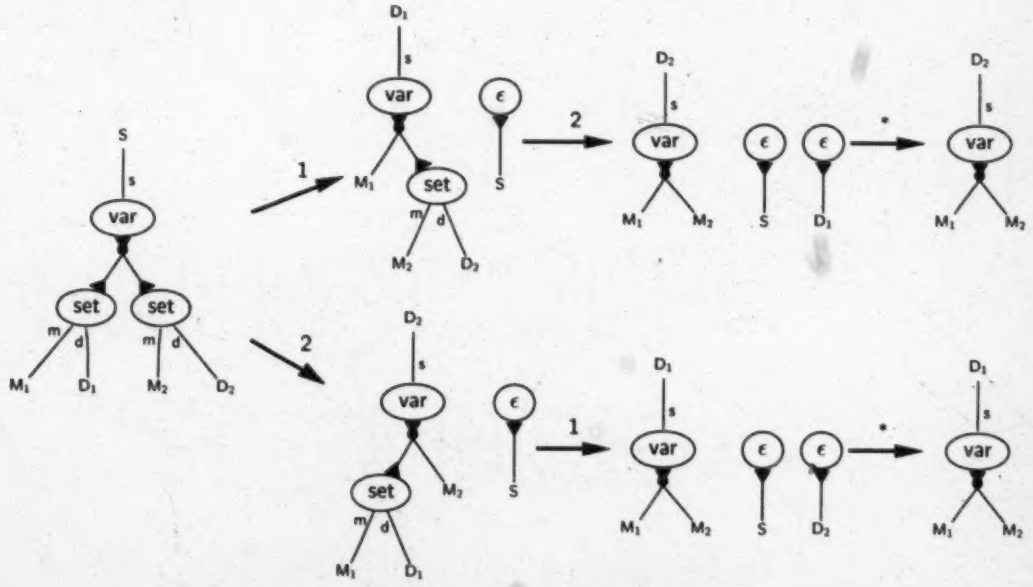
A variable (reference) can be defined as follows. The variable node var accepts requests on its principal (message) port, represented as nodes get/set. The variable also stores a private *state* at its s port. The request nodes get and set have a *data* port d that receives or carries the data item, and a *message* port m for the next message to the same "object". (The eraser ϵ and duplicator δ are standard IN nodes introduced in §2.2.) [⋈ 3.17]



⋈ 3.17: A Variable var has State and Responds to get/set Requests.

The get request is handled by duplicating the state S using the duplicator δ, returning the second copy to the required data port D and reconnecting the first copy to the variable. The set request is handled by erasing the old state S using an eraser ϵ and reconnecting the new state D to the variable. For both requests, the variable now listens to the rest of the message stream M.

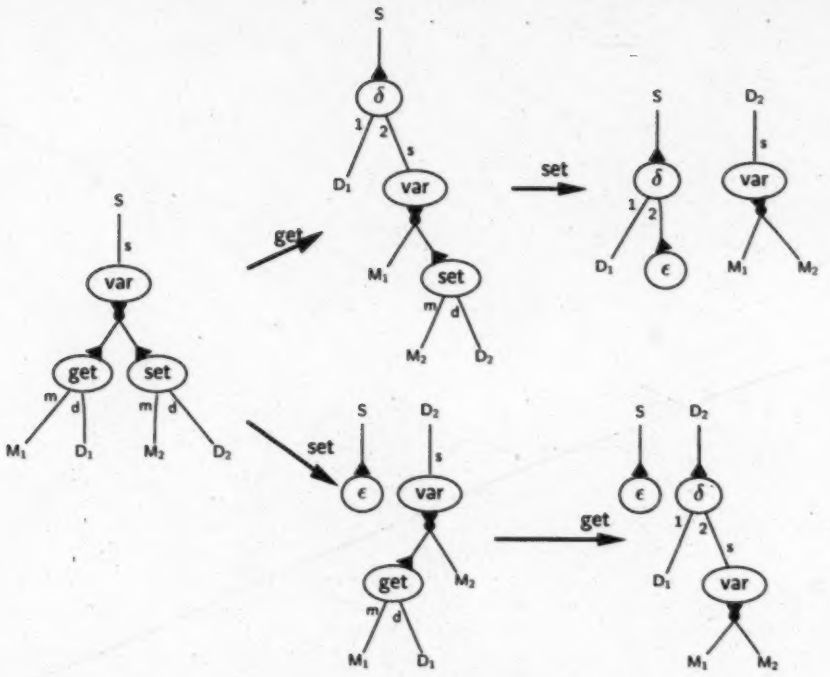
As expected, the evolution of a var that has two competing requests may depend on the order in which these requests are served. [⋈ 3.18]



⋈ 3.18: Outcome Depends on Order of Interaction.

On the other hand, there is no need to control atomicity of data transfer in this case. There is no danger of the two data items getting mixed up, because only references are passed around, not the actual data items.

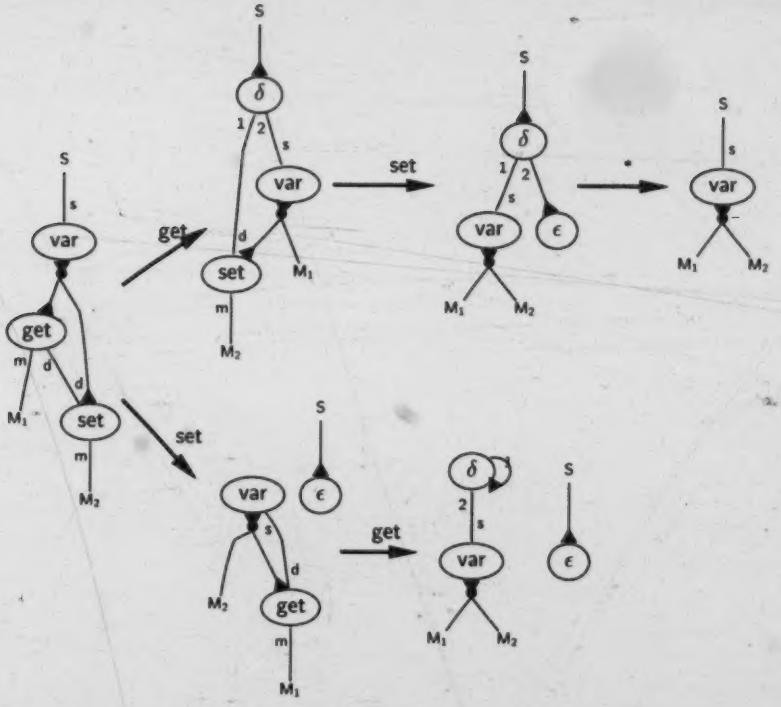
Now let's see what happens if we send get and set requests "simultaneously". The handling of get involves copying the data item, so conceivably some confusion due to non-atomicity could arise. [⋈ 3.19]



⋈ 3.19: get/set Interact as Expected.

Here the result that get returns depends on when the get request is processed, relative to the set request. But again there is no possibility of confusion of the data items: in the first case D_1 gets a copy of S (the other copy of S is erased), in the second case D_1 gets a copy of D_2 (and S is erased). In both cases the var is set to D_2.

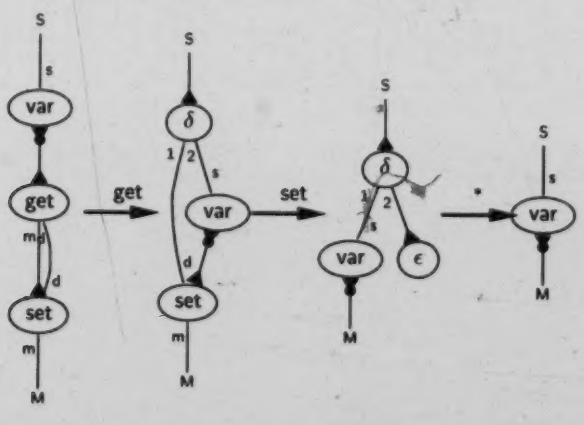
Now let's consider a negative example. Imagine that we want to get a value from a var, then set it to that same value (effectively, a null operation). We will try to simply short-circuit the data ports of get and set. [⋈ 3.20]



⋈ 3.20: The Lower Reduction Leads to a Vicious Circle.

If get is selected for reduction before set then the net reduces as expected. Unfortunately, the opposite case leads to a *vicious circle*, in which δ can never reduce, and therefore the new value is undetermined (the data item is "broken"). The reason is that we have failed to properly synchronize (sequentialize) the two requests. Our informal description stated that get will be executed before set, but our net fails to formalize this.

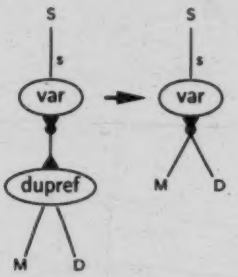
The fix is simple, we should enqueue set after get: [⋈ 3.21]



⋈ 3.21: The Correct Way to Sequentialize Requests.

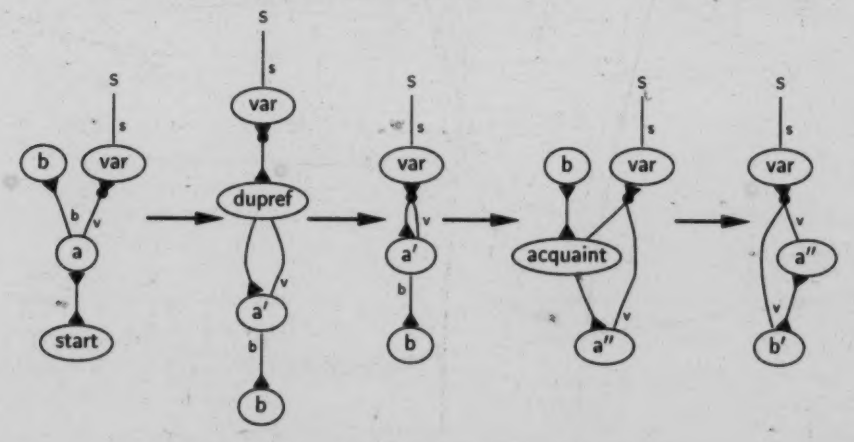
In order for a client of a var to be able to create a new reference to it and share it with others,

we introduce a message **dupref**, which creates an additional reference to **var**'s multiport instead of copying it. [⋈ 3.22]



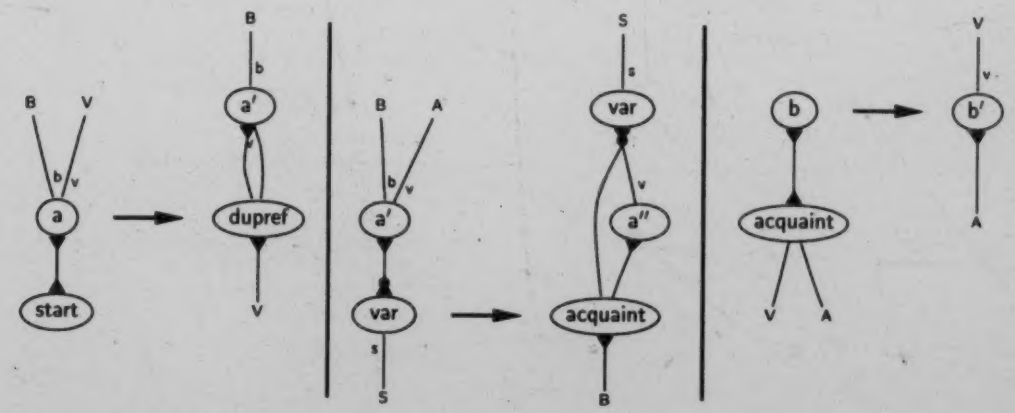
⋈ 3.22: Duplicating a **var** Reference.

Now consider this example: let's assume that a node **a** has a reference to a **var**, and wants to pass a copy to another node **b**. Then **a** could use **dupref** to get a second reference, and then a "private" message **acquaint** to send the reference to **b**: [⋈ 3.23]



⋈ 3.23: Example of Duplicating a **var** Reference.

Above we have assumed that **a** and **b** go through a simple sequence of steps, for which we need the following rules: [⋈ 3.24]

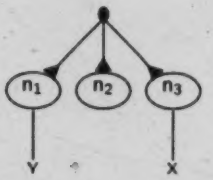


⋈ 3.24: Additional Rules for the acquaint Example.

For a similar task, Fernández and Mackie (1996a) introduce some separate new entities (*memory* and pseudo-edges), and only manage the access to the memory with conventional INs. We think that our approach stays truer to the spirit of conventional INs.

3.4 IN with Multi-Connections (INMC)

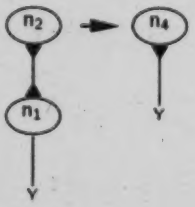
A property of conventional INs is that every edge connects exactly two ports. We may consider IN with Multi-Connections (INMC), or hyper-edges (in the graph-theoretical sense). These are connections between more than two ports. This would be useful in modeling hardware bus systems or software broadcasting systems, where several entities are connected to a shared communication medium. We denote a multi-connection with a bold black dot (connector) to which all participating ports are linked. [⋈ 3.25]



⋈ 3.25: A Sample INMC.

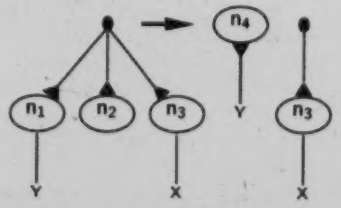
The interaction rules of INMC are of the same kind as the rules of conventional IN, but their application is a bit more involved. If more than one principal ports are involved in a multi-connection, then the choice of which pair interacts is non-deterministic. That pair of principal ports, their links and their nodes are excised from the net and replaced with the RHS of the rule. However, the connector and the remaining links to it (if any) are preserved. (If no links remain then we excise the connector as well.)

For example, given the rule [⋈ 3.26]



⋈ 3.26: INMC Rules are Like Conventional IN Rules.

the following reduction may occur: [⋈ 3.27]

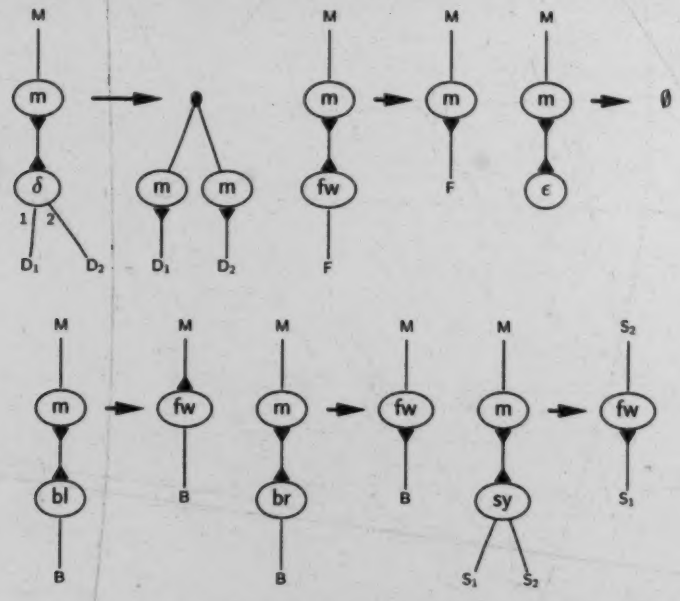


⋈ 3.27: A Sample INMC Reduction.

As you can see, reduction may lead to dangling links (links connected to only one port), which is contrary to the "spirit" of conventional INs.

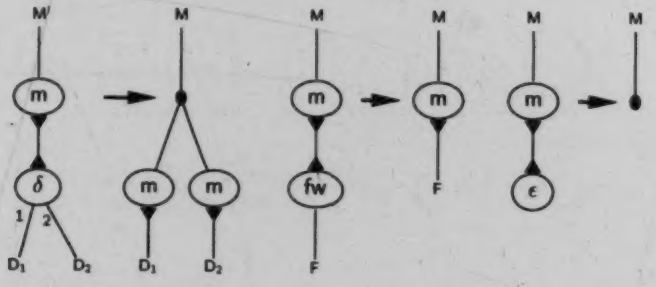
3.4.1 Example: Concurrent Combinator Process Graphs

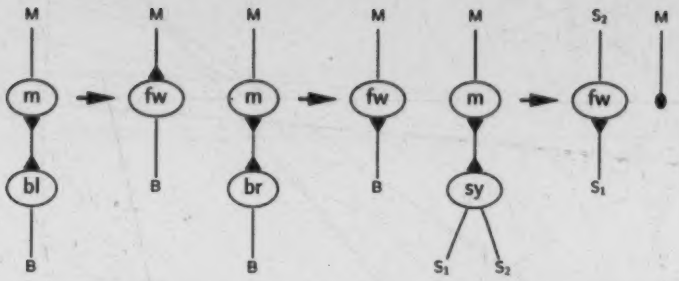
The *Concurrent Combinators* (used to represent the π-calculus without replication and choice in a finitary framework) of Honda and Yoshida (1994a) and the *Process Graphs* (PG) of Yoshida (1994) are similar to INMC, but in PG no shortcut edges are allowed in the RHS of rules. The *cc* system consists of 7 node types and the following rules: [⋈ 3.28]



⋈ 3.28: *cc* Process Graphs.

In order to make this system more similar to INs, we amend the above rules slightly by showing explicitly the fate of the M edge which is omitted in the first, third and sixth rules above. Of course, this does not change the meaning of the rules. [⋈ 3.29]





⋈ 3.29: Example of an INMC System: *cc* Process Graphs.

We now see that *cc* is an INMC system.

Chapter 4

Inter-representation of Models of Non-Determinism

It is not obvious which of the fore-mentioned ways of introducing non-determinism in IN should be chosen as "canonical". Therefore, a natural and interesting question is to see which of these models of non-determinism can capture which others, and at what price. This will allow us to compare the expressiveness of the models.

4.1 Introduction

This section presents some constructions and some partial results about the inter-representability of models of non-determinism for IN. A translation from a source system S to a target system T (where S and T are instances of two possibly different classes of IN, *e.g.* $S \in$ INMR and $T \in$ INMPP) consists of the following components:

1. For every node type $\mathsf{n} \in S$, a corresponding net $N \in T$ having the same free ports as n. We denote that N is the translation of n in this way: $N = [\![\mathsf{n}]\!]$.
2. For every rule $R \in S$, a corresponding set $[\![R]\!]$ of rules in T.
3. Possibly, a set X of extra (auxiliary) rules in T.

Definition 4.1.0.1 (Parallel Composition) *We say that an IN or INMR net* P *is the* parallel composition *of two disjoint*[1] *nets* M *and* N *connected along a certain interface (multiset of ports)* I*, iff:*

- The interfaces (multisets of free ports) of both nets are supersets of I: $i(\mathsf{M}) \supset \mathsf{I}$, $i(\mathsf{N}) \supset \mathsf{I}$, and furthermore $\mathsf{I} = i(\mathsf{M}) \cap i(\mathsf{N})$.
- The resulting set of nodes $n(\mathsf{P})$ is the disjoint union of the nodes of M and N: $n(\mathsf{P}) = n(\mathsf{M}) + n(\mathsf{N})$.
- The set of edges $e(\mathsf{P})$ is

$$e(\mathsf{P}) = e(\mathsf{M}) + e(\mathsf{N}) + \sum_{p \in I} \{p_{\mathsf{M}} - p_{\mathsf{N}}\}$$

  where p_{M} is the node port in M to which the free port p is connected, and similarly for p_{N}. Visually, we are taking the two "half-edges" $p_{\mathsf{M}} - p$ and $p - p_{\mathsf{N}}$, and joining them at the mid-point.

[1] Having no nodes or edges in common.

- The resulting interface is $i(\mathsf{P}) = i(\mathsf{M}) \cup i(\mathsf{N}) - I$, where minus denotes multiset difference.

We denote this symbolically as $\mathsf{P} = \mathsf{M}, \mathsf{N}$, where the interface I is often understood from the context or is not important.

For non-linear nets (not in the classes IN nor INMR), the definition above needs to be elaborated. For example, for INMP the interface I can include non-free ports (provided they are multiports), and the resulting interface $i(\mathsf{P})$ will include every multiport of M and N, even if some of them were in I.

Since item 1 above requires translations to work node-by-node, all our translations will satisfy the following very natural criterion:

Definition 4.1.0.2 (Uniformity of Translation) *A translation $[\cdot]$ is called* uniform *if it commutes with parallel composition (the translation of a parallel composition is the parallel composition of the translations of the components, along the same interface). Symbolically,* $[\mathsf{M}, \mathsf{N}] = [\mathsf{M}], [\mathsf{N}]$.

In addition, it is often useful to require of translations to satisfy the following property:

Definition 4.1.0.3 (Preservation of Interface Type) *The translation of a net with $p + a$ free ports (p principal and a auxiliary) is a net with $p + a$ free ports (p principal and a auxiliary).*

This together with the previous property allows us to conclude that every node is translated to a net with the same free ports, where the types (principal/auxiliary) of the free ports are also preserved.

4.1.1 Completeness and Soundness

Single-step reductions $M \to N$ in S are emulated by multi-step reductions $[M] \Rightarrow [N]$ in T. For this emulation to be faithful, we need two criteria which, following the mathematical logic tradition, are called *completeness* and and *soundness*.

Completeness means that T should not miss any reductions from S. This is easy to express: for every single-step reduction $M \to N$ in S, there should exist a multi-step reduction $[M] \Rightarrow [N]$ in T.

Soundness means that T should not introduce extraneous ("garbage") reductions that are not related to any reduction in S. This notion is harder to express, because it is not obvious how to define the notion of a target reduction being related to a source reduction. Please see (Nestmann and Pierce, 1996) for an in-depth discussion of this topic.

Finer Granularity We cannot hope to achieve the following: for every target reduction $[M] \Rightarrow P$ from a translation $[M]$ to an arbitrary net P, there exists a source reduction $M \to N$ such that $P = [N]$. The reason is that since the target reduction has *finer granularity* than the source reduction, it may be *partial*: it may need some more steps before it completes to a reduction that corresponds to a target reduction. Therefore, we need to relax our soundness criterion thus: for every source reduction $[M] \Rightarrow P$ there exists a "completing reduction" $P \Rightarrow [N]$ such that $M \to N$.

Interleaving Above we have stated that the target reduction can be completed to a *single-step* source reduction $M \to N$. However, this may not always be possible, because the partial target reduction $[M] \Rightarrow P$ may *interlace* parts from several independent source reductions. That is why we need to relax our criterion even further, by stating that the corresponding source reduction may be multi-step: $M \Rightarrow N$.

Auxiliary Pre-Commitment Steps Often the target system T will include auxiliary rules X which do not correspond to rules of the source system. Some of these can be "cleared up" by the completing reduction, but not all; we call such steps *pre-commitment* steps (from the explanations and example below you will achieve an intuition about such steps). If $[M] \Rightarrow P$ has used one of these steps, then it may be impossible to complete the

reduction to one corresponding to a source reduction, if we limit ourselves to the net $[\![M]\!]$. It should however be possible to do the completion if $[\![M]\!]$ is placed in an appropriate context, in which the auxiliary rule can complete the task that it was introduced for. Therefore, we can define soundness thus:

> For every target reduction $[\![M]\!] \Rightarrow P$, there exists a net C (context) and a completing reduction $P, [\![C]\!] \Rightarrow [\![N]\!]$ such that $M, C \Rightarrow N$.

By the locality of IN interaction, we know that $[\![M]\!] \Rightarrow P$ implies $[\![M]\!], [\![C]\!] \Rightarrow P, [\![C]\!]$ (the parallel component $[\![C]\!]$ cannot decrease the interaction abilities of its neightbors if it sits quietly on the side and does nothing). By the uniformity of our translations, we know that $[\![M]\!], [\![C]\!] = [\![M, C]\!]$. Therefore, $[\![M, C]\!] \Rightarrow P, [\![C]\!]$, and when we append the completing reduction to the end of it, we get $[\![M, C]\!] \Rightarrow [\![N]\!]$, which corresponds 1-to-1 to the source reduction $M, C \Rightarrow N$.

Appropriate Contexts Not all contexts C above make good "complements" of M, suitable to exhibit the soundness of the reductions of $[\![M]\!]$. There are "bad" contexts that will make almost any reduction of $[\![M, C]\!]$ appear sound, even if the reduction is totally unreasonable. For example, consider a context C consisting of erasers ϵ connected to all free ports of $[\![M]\!]$. The final result of the reduction of $[\![M, C]\!]$ could be an empty net, and any reduction of $[\![M]\!]$, even unreasonable ones, will lead to the same result. If we follow this thought, we quickly come to notions like ***bi-simulations*** for the π-calculus (Milner *et al.*, 1992), which are not yet available for IN. The development of behavioral equivalences (including bisimulations) for IN and non-deterministic IN is an interesting area for future research, and is outside the scope of this thesis.

Luckily, in §5.5.1 on the soundness of our translation of the π-calculus to non-deterministic INs, we don't need to resort to such heavy-duty contextual-testing techniques. We use a simple technique we call ***un-commitment*** that removes auxiliary pre-commitments in a deterministic way, and allows us to evaluate the "goodness" of a reduction of $[\![M]\!]$.

We give more formal definitions in §5.5.1. In this chapter we don't prove formally the faithfulness of our translations, but we introduce them gradually and give comprehensive examples, so we hope that we have made their faithfulness believable.

Pre-Commitment Steps: Example

To clarify the point about pre-commitment steps, consider the following example, inspired by the "blocks world" of AI. We have two marked spots on a table, and no more than two glasses. The valid configurations of our world have glass at some of the spots, and no glasses elsewhere. Let's assume that a man S wants to switch the places of two glasses A and B placed on the spots. He could pick the glasses with two hands, cross his hands, then put them back on the table while reversing their positions.

Let's further assume that a single-handed[2] man T wants to emulate S. He could do that by picking up the glass A, putting it somewhere on the table (out of a spot), moving the other glass B to the empty spot, then placing A at the spot that was just freed by B. If our unfortunate man T was also blind, and someone had stolen the glass B, then T might not notice the missing glass, and would be stuck in the middle of his operation. Then T will place the world in an invalid configuration. The man with two hands S, even if blind, would not have this problem, since he would feel the missing glass in the very beginning of his operation.

Of course, there are several ways for T to correct the situation. One is to return A to its original spot. However, if he had a rule to do that, and the rule was not conditioned on the lack of a glass

[2] We considered using the term "mono-dextrous", but dismissed it as unacceptably politically-correct.

B,[3] then he may well go into an infinite (diverging) loop of moving A back and forth. Another corrective action would be to move A to the empty spot, but we would be changing the rules of the game (the algorithm).

Therefore, given a correct but "incomplete" configuration with only one glass A, our man T will get stuck in an incorrect configuration where A is not on a spot. But if we complete the configuration by adding a glass B (a "context"), the incorrect configuration can be transitioned to a correct one.

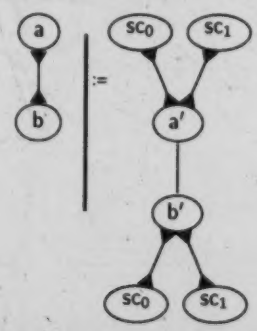
4.1.2 Example: Translating INMR to INMPP

Before going deep in inter-representation translations, let's consider an example. We have an INMR system consisting of two nullary node types a and b, and two rules $\mathsf{a} \bowtie \mathsf{b} \to N_0$ (R_0) and $\mathsf{a} \bowtie \mathsf{b} \to N_1$ (R_1) where $N_0 = \mathsf{a} \bowtie \mathsf{b}$ and $N_1 = \emptyset$ are two different nets. We want to represent this in INMPP.

INMPP doesn't have multiple rules per redex, so we need to emulate this somehow. In INMR, the system "decides" whether to apply R_0 or R_1, but in INMPP the nodes themselves would have to make that decision (the decision must be "internalized" into the nodes). So we need target node types a′ and b′ that are capable of "deciding" which rule should apply.

One way is to introduce *self-commitment* nodes sc_0 and sc_1, which would commit a′ and b′ to *committed* ("decided") nodes a'_i and b'_i ($i = 0, 1$). Of course, when two committed nodes meet, their decisions better be compatible: if a'_i wants to apply rule R_0 and b'_j wants to apply rule R_1, then we are in trouble. For this example, we can easily go around this complication by positing that $\mathsf{a}'_i \bowtie \mathsf{b}'_j$ should reduce by rule R_k, where $k = (i + j)\%2$ (% is the modulus operation).

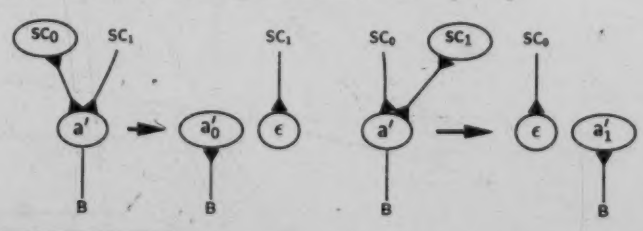
Therefore we translate the source net $\mathsf{a} \bowtie \mathsf{b}$ as follows: [⋈ 4.1]



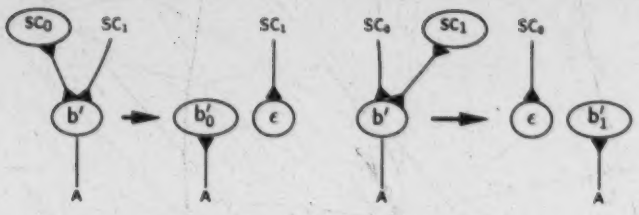
⋈ 4.1: Translating INMR to INMPP Example.

It is obvious that our translation is uniform (translates node-by-node).

We need extra (auxiliary) rules to express commitment: [⋈ 4.2]



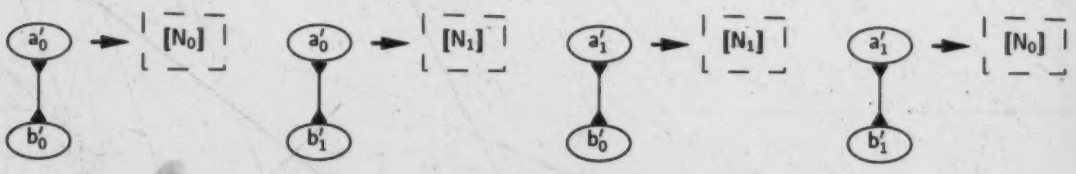
[3] IN are notoriously bad at checking whether something is *missing*. The moment a single-principal-port node n turns its attention (principal port) in a certain direction, it cannot move it elsewhere without external help. If there was nothing in that direction, n would be stuck forever.



⋈ 4.2: INMR to INMPP Example: Auxiliary Rules.

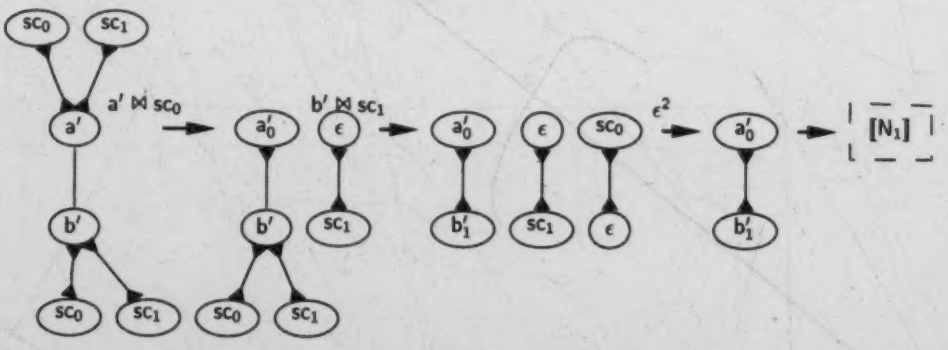
To illustrate the first rule: a' becomes a committed node a'_0, and both self-commitment nodes of a' are dropped (sc_0 is consumed by this rule, while sc_1 will be killed by ϵ).

Finally, we need 4 main rules that directly emulate source rules: [⋈ 4.3]



⋈ 4.3: INMR to INMPP Example: Main Rules.

An example of a reduction is given below: [⋈ 4.4]



⋈ 4.4: INMR to INMPP Example: A Sample Reduction.

We can also illustrate the fine point about soundness that we raised at the end of §4.1 and before the pre-commitment steps example. Consider a source net consisting of a single a node. Its translation is the "excited" configuration consisting of three nodes a', sc_0, sc_1. It reduces to a target net consisting of a single committed node a'_i, however that target net is *not* the translation of any source net. This unfortunate target net cannot reduce any further. Only if we put it in an appropriate context (one having **[b]** connected to **[a]**) will it be able to "prove" that it is good. (Not every context will lead to a good net. For example, a context containing another unconnected node will not do.)

Note This translation is very similar to the INMR→INMPP translation we develop in §4.1.2, but there we are more formal and our exposition has lesser narrative (pedagogical) value.

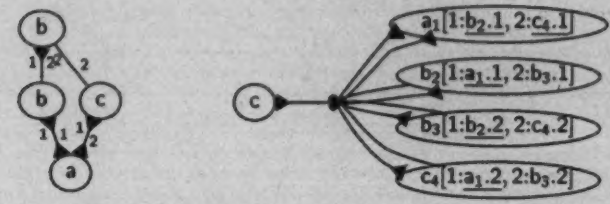
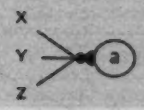
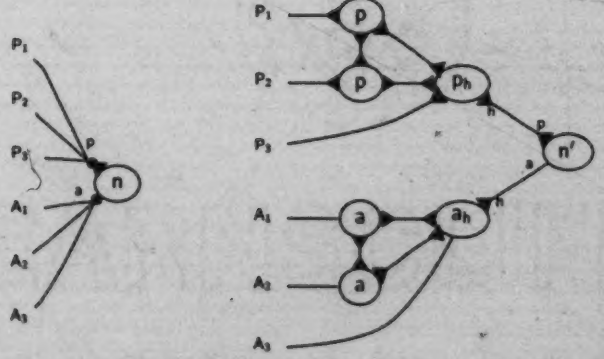
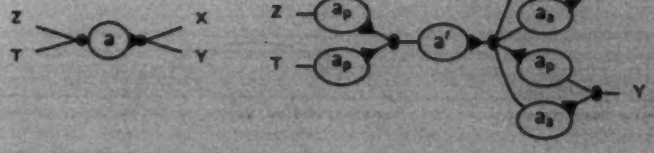
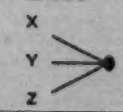
4.1.3 Basic Ideas

We summarize some basic ideas for translations in the following table. We develop some of the translations with varying degree of detail in subsequent sections. This table should be used as a

"quick reference" to the different transaltions; the reader likely will not be able to understand the ideas from it alone.

If the **Section** column shows **??**, then the corresponding translation is not developed in detail in this thesis, but is only hinted upon. If the **Target** column shows —, then a faithful translation is not possible.

Source	Target	Section/Explanation
INMR	INMPP	§4.1.2 Use *self-commitment* nodes sc_i to "decide" for a' which rule to apply, by committing a' to one of several alternatives.
INMR	INMP	§4.5 Use *self-commitment* nodes (same as for INMR→INMPP).
INMR	INMC	§?? Use *self-commitment* nodes (same as for INMR→INMPP).
INMPP	INMR	§4.2 No faithful translation is possible.
INMPP	INMP	§4.6 Represent the n principal ports $x \ldots y$ as one principal multiport of invariant arity n. Use *markers* $a_x \ldots a_y$ to distinguish the edges stemming from the multiport.
INMPP	INMC	§4.3 Represent every principal port of a explicitly as a pair ("diamond") of two nodes, *passive* a_p and *active* a_a (similar to the INMP→INMC translation of §4.4).

Source	Target	Section/Explanation
INMPP	INMC	§4.3.1
		Collapse all links into a central global connection point. Capture all information about a node's immediate neighbors in the node type (this requires an infinite number of node types). We call this below the ***grotesque translation***.
INMP	INMR	§4.2
	—	No faithful translation is possible.
INMP	INMPP	§??
		Represent every multiport explicitly, as a "ring" of inter-connected multiport nodes: ***principal*** p or ***auxiliary*** a. The nodes closest to the ***host*** n' (p_h and a_h) can forward instructions from other ring members to the host (*e.g.* when an interaction with P_i has occurred). They can also take instructions from the host (*e.g.* when it wants all multiport nodes to change their types from principal to auxiliary or vice versa), and propagate them to the other ring members.
INMP	INMC	§4.4
		Represent every edge to a multiport explicitly. An edge to an auxiliary multiport of a is represented as a ***passive*** node a_p. An edge to a principal multiport of a is represented as a pair ("diamond") of two nodes, ***passive*** a_p and ***active*** a_a.
INMC	INMR	§4.2
	—	No faithful translation is possible.

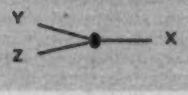
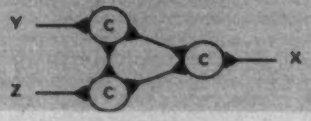
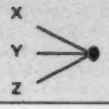
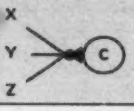
Source	Target	Section/Explanation
INMC 	INMPP 	§?? Represent a connection point by a "ring" of *connector* nodes c.
INMC 	INMP 	§4.7 Represent a connection point by an explicit *connector* node c.

Table 4.1: Inter-representation of Non-Determinism: Basic Ideas.

Of course, we could limit our study to only a linear number of translations (for example, INMR→INMPP→INMP→INMC→INMPP), and then define other translations as compositions of these basic ones. However, this would lead to an explosion of complexity (and cost) for the composite translations, and would not give us additional insight into the relative expressive power of the systems under consideration. That is why we don't view such composite translations as very useful, and do not pursue them.

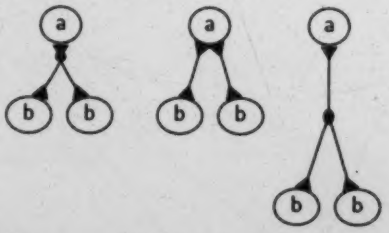
4.2 Separation Result: INMR is Weaker Than the Rest

We start with a negative (separation) result: it is not possible to represent any of the other systems in INMR faithfully. We will pose a natural definition of "reasonable" and "faithful" translation as we go along with the proof of this result. Thus, INMR are less expressive than all the other systems. The following concept is not representable in INMR:

Concept 4.2.0.1 (Simultaneous Attentiveness) *A multi-interaction node can be* simultaneously attentive *towards more than one edge. This means that the node can interact with a principal port appearing on any one of the edges.*

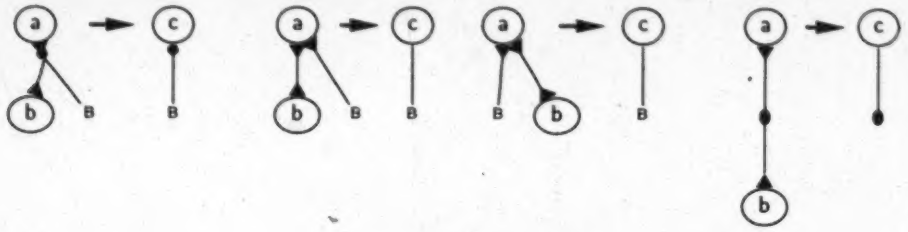
Since INMR are structurally the same as IN (single principal port per node), they don't contain simultaneously attentive nodes. However, a net can be simultaneously attentive, because a net can have more than one principal port. Since the translation of a node of the source system can be a whole net of the target INMR system, this gives us simultaneous attentiveness. But if a net is simultaneously attentive towards two edges, then it can be split into two parts which are *not* attentive towards each other, and therefore cannot act as a coherent whole.

To make this more precise, we will construct a counter-example net in each of the three target systems INMP, INMPP and INMC: [⋈ 4.5]



⋈ 4.5: INMR Counter-Example in INMP, INMPP and INMC.

We consider the following interaction rules governing a and b in the respective three systems (the middle two rules are both for INMPP). [⋈ 4.6]

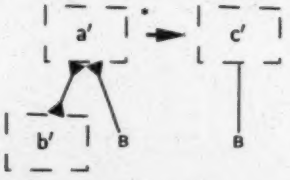


⋈ 4.6: Counter-Example Rules

It is an important point that the node of c is not principal (we use that below). If a node with no principal port doesn't make sense to you, please assume that a has an extra auxiliary free port on the top, and that c has an extra principal port on the top, corresponding to a's free port.

Let's assume that the above counter-examples are representable in INMR. Then the configuration consisting of the node a and two outgoing edges (for all source systems INMP, INMPP, INMC) must be representable as a net a′ with two free ports, both of them principal. Similarly, the translation of b must be a net b′ with a single free port which is principal, and similarly about c. The cut a ⋈ b must be represented by a "cut" between the two nets a′ ⋈ b′, in other words there should be a single edge between the two nets, connecting principal ports. All this follows from the uniformity (4.1.0.2) and preservation of interface types (4.1.0.3) that we want our translations to satisfy.

The translated nets a′, b′, c′ are shown on the left in the next figure. Furthermore, the interaction rules of ⋈ 4.6 must be representable by a reduction like the one shown on the right: [⋈ 4.7]



⋈ 4.7: Hypothetical translation of the counter-examples in INMR.

Here the free port B of c′ is auxiliary, like the corresponding port on ⋈ 4.6.

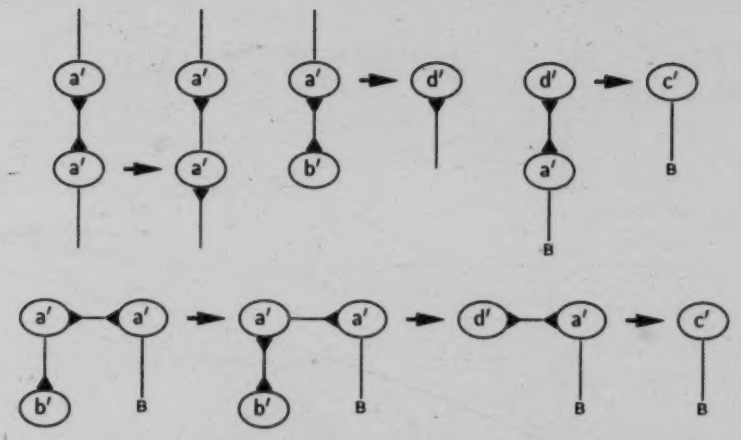
Theorem 4.2.0.2 *If the net a′ does not have vicious circles, then the reduction ⋈ 4.7 is impossible in IN and INMR.*

Proof We use a standard structural result about IN, the "combing of a net" §2.3.4. The same result holds both about IN and INMR, since they are structurally the same. Let's comb a′ ⋈ b′ and then consider the tree of the root B (call it T). It is connected to the rest of a′ ⋈ b′ (call it R) by auxiliary (unoriented) edges only. Therefore R cannot affect T, no matter what is the reduction, and thus cannot change the B port from principal (in a′ ⋈ b′) to auxiliary (in c′).

In other words, we find an auxiliary split (split across auxiliary edges only) that separates B from the active part of the net a′ ⋈ b′, and so this active part cannot affect B. ◊

Lest you think that INMR are completely useless as a model of non-determinism for IN, we will now present a set of INMR rules that *can* implement both of the INMPP reductions of ⋈ 4.6, but in a rather peculiar way; please read on.

Let's postulate the interaction rules given on the top row of the next figure. [⋈ 4.8]



⋈ 4.8: INMR can implement both INMPP reductions.

Given the above rules, if we postulate that **[a]** = a′ ⋈ a′, **[b]** = b′ and **[c]** = c′, then the configuration **[a** ⋈ **b]** on the bottom left of the figure reduces to **[c]**. This implements one of the INMPP reductions. Furthermore, since none of the INMR rules single out any of the two free ports of **[a]**, the same rules also implement the other INMPP reduction.

What is the trick here? The trick consists of several parts:

1. **[a]** is a non-reduced net (contains the cut a′ ⋈ a′). Normally this is useless in IN since IN are confluent, and so usually one postulates as an optimization that translations (and RHS of rules) should be reduced nets. But this is not the case for INMR.

2. The rule for a′ ⋈ a′ is asymmetric, therefore essentially-INMR. What we really have here are *two* rules, one as presented above, and the other one having the mirror image of the RHS.

3. **[a]** does not respect the port types of a. Instead of having two free principal ports, it has auxiliary ports, but the internal contents of the net can activate *either one* of these ports (make it principal). To restate, although **[a]** doesn't have the ability to interact *immediately* on any of its free ports, the rule for a′ ⋈ a′ can make either one of them active by *internal reductions*.

Because of the last item above, **[a]** violates the assumption from Definition 4.1.0.2 that translations should respect the types of free ports. This assumption is pretty natural; typically it is part of π-calculus bisimulation relations (*e.g.* Honda and Yoshida (1994a) call it the *action predicate*). For IN it is also the only sensible alternative, since:

- If a free principal port p is translated to an auxiliary port a in a net N, then the net is not ready to interact on a, so the translation is not faithful. Furthermore, if N is reduced (which is a harmless assumption in IN), then it cannot change the type of a (activate it) by internal reductions only.

- If a free auxiliary port a is translated to a principal port p in a net N, then N can never change p's type, so the mismatch will exist forever.

However as we saw above, INMR nets can change the types of free ports, provided that inside the net there are *trigger* cuts. If the rules for these cuts are essentially-INMR, then the net can change port types in several different ways, and thus implement several different activation patterns. Above we modeled INMPP "and-parallelism" of ports (both ports are simultaneously active) with INMR "or-parallelism" (the net can activate either port, but not both at the same time).

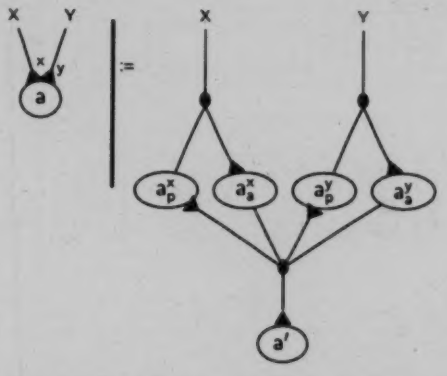
Therefore it is not obvious that our assumption that translations should preserve free port types is the "right" one. If we were able to model INMPP with INMR "to our satisfaction", we would have forfeited this assumption. However, we were not able to do so. There is a significant problem with the rule ⋈ 4.8 above: the trigger cut makes a choice/commitment *blindly*, and may well choose wrong. Namely, if a′ turns its principal port not towards b′ which is ready to interact but towards the free port B, no interaction will be possible. We have presented only the successful path, but there are also failing paths. INMR do offer or-parallelism, but offer no way to control it.

Worse yet, commitments are irreversible. Once a′ has turned its principal port towards B, there is no way for the net to "get its attention back" and correct a wrong commitment.

Commitments/activation are not "renewable" either. We are able to construct something like a token-ring network: make a vicious circle of a′ nodes, then turn one of them around. Postulate that the resulting cut is the *token*, and then the a′ ⋈ a′ rule can move it around the ring (this particular token may blunder back and forth instead of rotating around the ring, but it's also possible to construct a rotating token). However, we are unable to extract the information about the position of the token out of the ring: once a ring member turns its attention outside of the ring in order to see whether it needs to report its state to another node (a client of the ring), the ring breaks apart. It might be possible to make the client of the ring return the ring to its rotating state by a further construction, but things become too complicated.

4.3 Representing INMPP as INMC: *Port Diamonds*

We can capture INMPP in INMC using the following basic idea: represent every principal port x of a explicitly as a pair ("diamond") of two nodes, *passive* a_p^x and *active* a_a^x. The active node is attentive towards the outside ("listens for incoming connections", in the parlance of TCP/IP network software). The passive node is attentive towards the *host* node a′, so that it can take instructions from it to passivate the other member of the pair should a′ evolve into a node which has an auxiliary port in that spot. [⋈ 4.9]



⋈ 4.9: Representing INMPP as INMC: the Basic Idea.

Since the representation is similar to the INMP→INMC translation of §4.4, we won't go into further detail here.

Instead, we present below another translation idea that shows both the power of INMC, and how far they have strayed from the "spirit" of conventional INs.

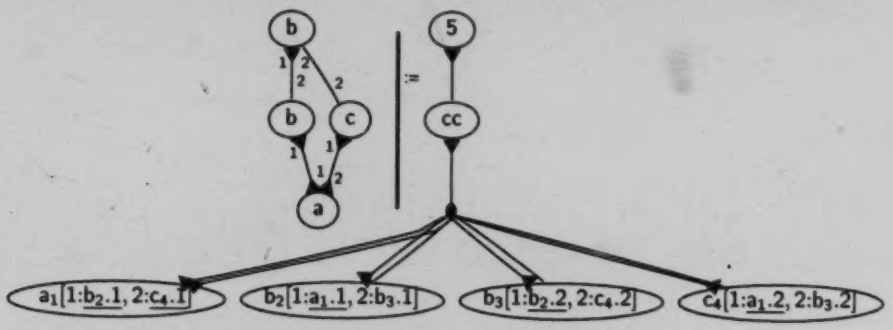
4.3.1 The Grotesque Translation

Another way in which we can represent INMPP in INMC is this: connect all nodes to a central node cc,[4] remove the inter-connections between the nodes themselves, and instead represent the linkage information inside "rich" node-types. We assume without loss of generality that all ports in the original system are numbered for easier identification. In order to represent the linkage information, we need the following:

Instance IDs These are globally unique natural number identifiers (we assume given the rules for builtin arithmetics §2.5.1). The center cc keeps the next globally-available ID N, and allocates IDs at runtime (after the initial net configuration starts reducing) to newly-created nodes. We denote instance-IDs below as a subscript to the original node type-name.

Instance Labels The type of every node is changed to reflect the local surroundings (neighboring nodes and connections) of the node. For example, a node of type b with ID=2, which is connected to nodes of types a and b, might get a new node type $b_2[1{:}\underline{a_1.1}, 2{:}b_3.1]$. This label further specifies what are the IDs of the neighboring nodes (a_1 and b_3), as well as which are the exact ports that implement the linkage (*e.g.* the b_3 node is connected through its port 1 to port 2 of the b_2 node. In other words, the label reflects an edge $b_2.2{-}b_3.1$.) We also underline the principal port(s) of the host node.

For example, the INMPP configuration on the left will be translated to the INMC configuration on the right. Note that the center cc holds the next globally-available ID, the number 5. [⋈ 4.10]



⋈ 4.10: Translating INMPP to INMC.

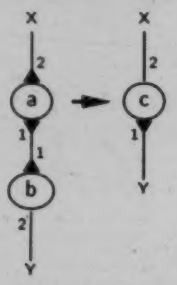
The application of a rule becomes quite an involved sequence of events:

1. The two nodes involved in a redex "find" each other and commit to the interaction.
2. Then they notify cc.
3. The center cc allocates sequentially new IDs for the nodes from the RHS of the rule.
4. Then cc removes the two nodes from the LHS of the rule (the redex), and inserts (instantiates) the nodes from the RHS.
5. Then cc notifies the existing nodes on the interface of the redex and the new nodes of their new surroundings. In effect, this establishes the linkage between the existing nodes and the new nodes.

Since the result of the translation and the nets capturing the intermediate steps of a reduction are very simple (degenerate), we don't represent them as figures, but as simple node sets, and write our rules in a textual notation.

[4] A "central committee".

Let's represent the following sample rule of the INMPP system we used as an example above. [⋈ 4.11]



⋈ 4.11: Sample INMPP Rule to Translate to INMC.

Below is an elaboration of the rule execution sequence we outlined above:

- For any IDs m and n and any labels $x_x.p$ and $y_y.q$, we postulate rules

$$\mathsf{a}_m[1{:}\underline{\mathsf{b}_n.1}, 2{:}x_x.p] \overset{C}{\bowtie} \mathsf{b}_n[1{:}\underline{\mathsf{a}_m.1}, 2{:}y_y.q] \quad \rightarrow \quad \mathsf{ab}_{m1n1}[1{:}x_x.p, 2{:}y_y.q] \bowtie C$$

  Here $\overset{C}{\bowtie}$ symbolizes that a and b interact through the same connection point to which cc is connected (C is a "variable" that matches cc). The node ab represents a commitment by a and b to the interaction. We have not indicated a principal port for this node in its label, and so we postulate that its principal port is towards the center C.

- The next set of rules specifies how does the center cc proceed to create the resulting node(s), in this case c:

$$\begin{aligned}\mathsf{ab}_{m1n1}[1{:}x_x.p, 2{:}y_y.q] \bowtie \mathsf{cc}(N) \quad \rightarrow \quad & \mathsf{c}_N[1{:}\underline{x_x.p}, 2{:}y_y.q] \bowtie \\ & \mathsf{cc}_{\mathsf{a}_m.1,\mathsf{b}_n.2}(N+1)[1{:}\underline{x_x.p}, 2{:}y_y.q, 3{:}c_N.1, 4{:}c_N.2]\end{aligned}$$

  Here c is instantiated with an ID of N (so we denote it c_N), and is immediately connected to the existing interface nodes x_x and y_y. The center cc goes into a transitionary state $\mathsf{cc}_{\mathsf{a},\mathsf{b}}$ in which it seeks to "reconnect" its ports 1–3 and 2–4.

- Finally, the transitionary state $\mathsf{cc}_{\mathsf{a},\mathsf{b}}$ proceeds to "reconnect" the existing nodes x_x and y_y to the new node c_N in two steps:

$$\begin{aligned}\mathsf{cc}_{\mathsf{a}_m.t,\mathsf{b}_n.u}[1{:}\underline{x_x.p}, 2{:}y_y.q, 3{:}z_z.r, 4{:}z_z.s] \bowtie x_x[\ldots p{:}a_m.t\ldots] \quad & \rightarrow \quad \mathsf{cc}_{\mathsf{b}_n.u}[1{:}\underline{y_y.q}, 2{:}z_z.s] \bowtie x_x[\ldots z_z.r\ldots] \\ \mathsf{cc}_{\mathsf{b}_n.u}[1{:}\underline{y_y.q}, 2{:}z_z.s] \bowtie y_y[\ldots q{:}b_n.u\ldots] \quad & \rightarrow \quad \mathsf{cc} \bowtie y_y[\ldots z_z.s\ldots]\end{aligned}$$

In the last step we have glossed over one issue: at the time when $\mathsf{cc}_{\mathsf{a},\mathsf{b}}$ wants to reconnect x_x and y_y to z_z, the nodes x_x and y_y may have already committed to interactions of their own. In that case it becomes quite complicated to figure out what are the correct new edges that have to be connected. We can easily avoid this complication by introducing an extra check before the first step, which ensures that nodes can only commit to an interaction while the center cc is in its primary state, not in the "reconnection" state $\mathsf{cc}_{\mathsf{a},\mathsf{b}}$.

Note Here we have used an enumerably-infinite number of node types and interaction rules. It should be possible to reduce this to a finite number of types, by attaching to every node the instance-IDs of connected nodes, in the form of counting sequences (similar to §4.7.1). However, we do not care to develop this refinement further.

Note The same representation idea should work, with some modifications, for a translation of INMP to INMC. Since in INMP a multiport may be connected to several nodes, we will need node types to be able to represent a ***multiset*** of instance-labels for every port.

What Have We Accomplished? We have named the construction of this section the *grotesque* translation for rather obvious reasons: it nullifies most of the useful aspects of INs:

- Rather than enjoying distributed local connections, all nodes are connected to a central entity.
- Interaction is also global. It depends on a central entity (the ID dispenser), which becomes a bottleneck and greatly limits parallelism.
- Rule applicability and execution becomes a complicated global procedure, rather than being decided by simple local principles.

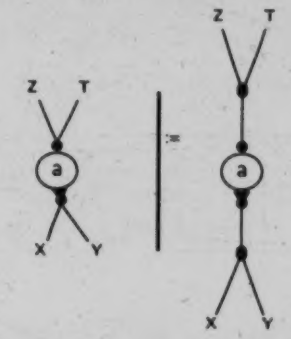
So what was our goal in presenting this translation? To show the power of INMC, even though this power is (in our opinion) obtained at the expense of straying far off from the positive aspects of conventional IN.

Please refer to §4.4 for a translation INMP→INMC; basically the same idea should work for INMPP→INMC to give us a "nice" translation..

4.4 Representing INMP as INMC: *Port Diamonds*

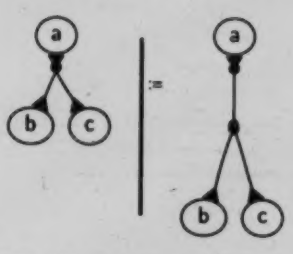
4.4.1 Basic Idea: Represent a Multiport as a Connection Point

We try to emulate INMP in INMC using the following intuition: move the INMP multiports away from the node boundary and into an INMC connection point. [⋈ 4.12]



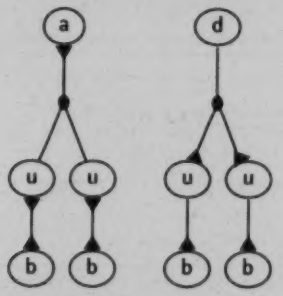
⋈ 4.12: Convert a Multiport to a Connection Point (Naive Approach).

This over-simplified approach doesn't work, since the connection point doesn't have any directionality. The multiport belongs to a, while its translation to a connection point is shared between a and all other nodes connected to a on the same multiport. (We now drop the auxiliary multiport of a from our discussion for a while, so as not to clutter the exposition.) As a counter-example, on the figure below b and c can't interact directly, while b and c can. This cannot lead to a behaviorally correct translation. [⋈ 4.13]



⋈ 4.13: The Naive Approach Doesn't Work.

Therefore, we introduce directionality by inserting *unidirectional* nodes u, which preserve the "orientation" of the multiport of a, *i.e.* the principal ports of u are turned towards b. It is "obvious" that if we are translating a node d with an *auxiliary* multiport, then its u nodes must be turned the other way around, which we also illustrate below. [⋈ 4.14]

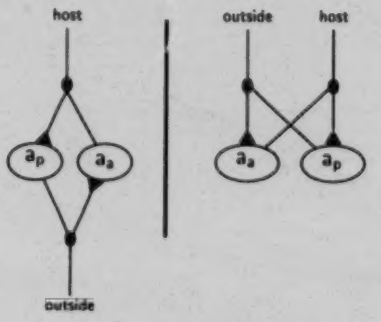


⋈ 4.14: Insert Unidirectional Nodes Which Point in the Same Direction as the Multiport.

There is still a problem here: imagine that the original INMP system has a rule a ⋈ b → c similar to the first one of ⋈ 4.6, a rule in which the principal multiport of a "becomes" an *auxiliary* multiport of c. In that case the interaction b ⋈ u and then b ⋈ a must *passivate* the other u node belonging to a, [5] in order to keep with the orientation we have illustrated for the u nodes of d.

4.4.2 Port Diamonds

The main challenge of the INMP→INMC translation is to develop a configuration representing a principal multiport, such that it can be attentive both towards the outside ("listen for incoming connections"), and towards its *host* node a, so that it can obey passivation commands from the host. We have found such a configuration, the *diamond*. It consists of a pair of nodes, *passive* a_p and *active* a_a.[6] [⋈ 4.15]



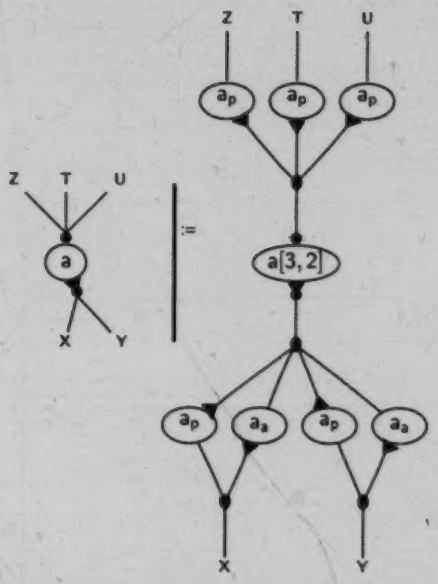
⋈ 4.15: The *Port Diamond* Configuration.

The diamond is a subtler configuration than it appears. Note that there is a ("negative") feedback loop between a_a and a_p. On the right side of the figure we have layed out the same configuration differently, to make its similarity to a flip-flop of electrical engineering more apparent.

The translation of a node a with multiports is: [⋈ 4.16]

[5] Turn the node u around, so that it looks towards the "outside world", in this case b, with its auxiliary (passive) port.

[6] For a host b, the two nodes in the diamond would be b_p and b_a. The active node must be dependent on the type of its host; we suspect that the passive node can be uniform (independent of the type of its host), but for simplicity we don't pursue this here.



⋈ 4.16: Translation of an INMP Node With Multiports.

Every edge to a multiport is represented explicitly. An edge to an auxiliary multiport is represented as a passive node. An edge to a principal multiport is represented as a diamond consisting of a passive and an active node. The host a knows the current arities of all its multiports (in the example above, 3 and 2 respectively). (This would require an enumerably infinite set of node types. It is possible to represent the arities with unary numbers instead, as we do in §4.7.1. For simplicity we do not develop this here.)

An interaction a ⋈ b is translated to the following sequence of steps:

1. a_a interacts with b (or b_a, in case the principal port of b is also a multiport). This interaction creates *pre-commitments* pre that are sent to both hosts a and b.
2. If a host (say a) approves the pre-commitment, it becomes a *pre-committed host* a′ and sends an *acknowledgment* ack_a to the other host. While in the pre-committed state, a′ won't approve any other pre-commitments.
3. If a host does not approve the pre-commitment, then it sends a *rejection* rej to the other host. This may happen because the host is in an inappropriate arity state (see Arity Constraints in §3.3.2), or is pre-committed to another interaction, or is even fully committed to another interaction and is in the process of implementing it.
4. If any of the hosts sends a rej, that rejection annuls any acknowledgment or annihilates any other rejection, and returns the port-diamond to its initial state.
5. If both hosts send ack's, then they become *fully-committed hosts* a″ and b″ and proceed with implementing the interaction. That includes:
6. The two hosts decrease their principal port arities (corresponding to removing the cut edge).
7. None, one, or both of the hosts passivate their principal port "diamond bundles", if the corresponding port on the RHS is auxiliary.
8. The hosts cooperate to instantiate the RHS of the corresponding rule and migrate all their ports/edges to that RHS instance.

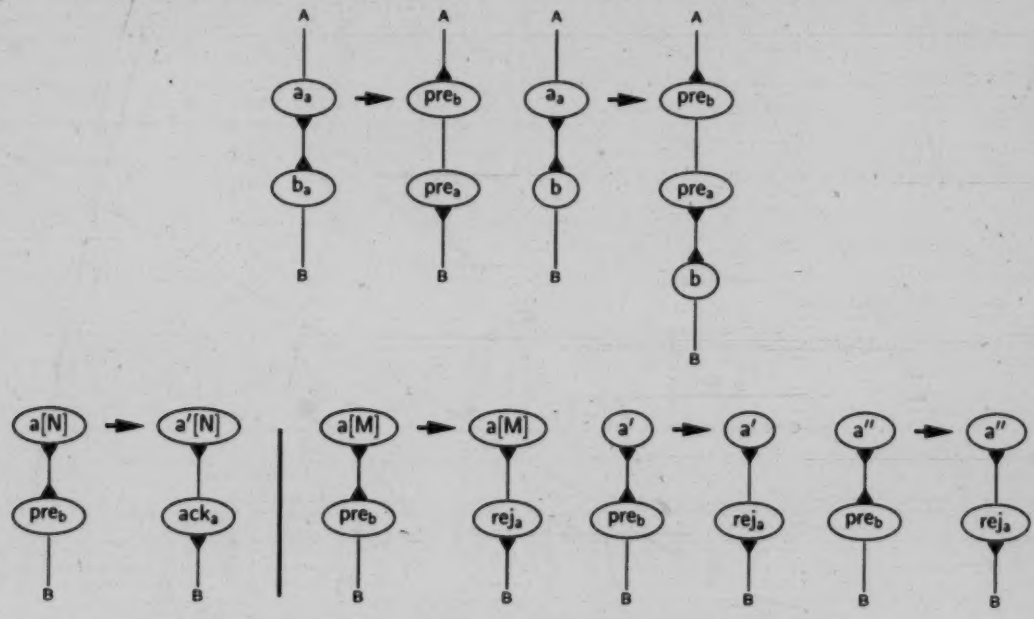
9. Finally, the two hosts disappear (corresponding to removing the cut nodes).

The sequence above is laborious, but not too complicated conceptually. Please note that certain important properties of the source system INMP have allowed us to keep the sequence relatively simple:

- The property of ***binary interaction*** makes the negotiation process 1–5 possible.
- The migration in step 8 is made easier by not allowing implicit merging of edge bundles on the RHS.

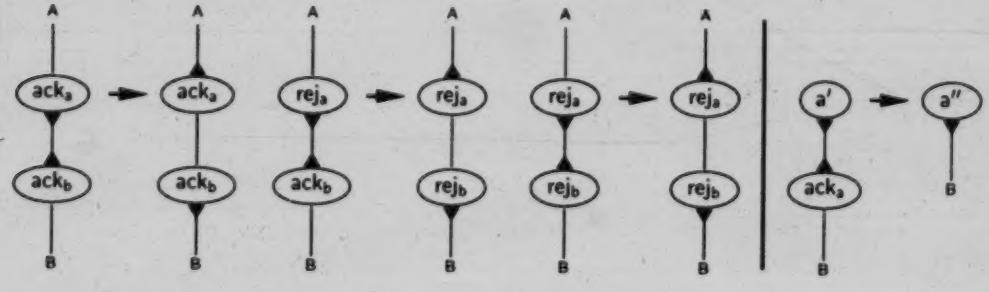
4.4.3 INMP→INMC Interaction Steps (Protocol)

We now give the interaction rules corresponding to the description above. First come the negotiation rules (pre-commitment, acknowledgment, rejection). The a ⋈ pre_b → ack rule is defined for those arity states a[N] that satisfy the arity constraints of the original a ⋈ b rule. All other arity states (notated with a[M]), as well as all non-initial host states, lead to rej. [⋈ 4.17]



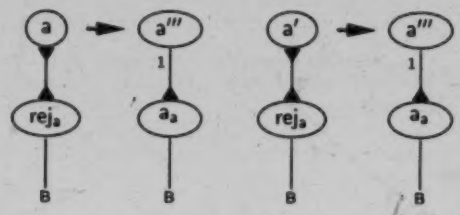
⋈ 4.17: Negotiation: Generation of Pre-commitment, Acknowledgment, Rejection.

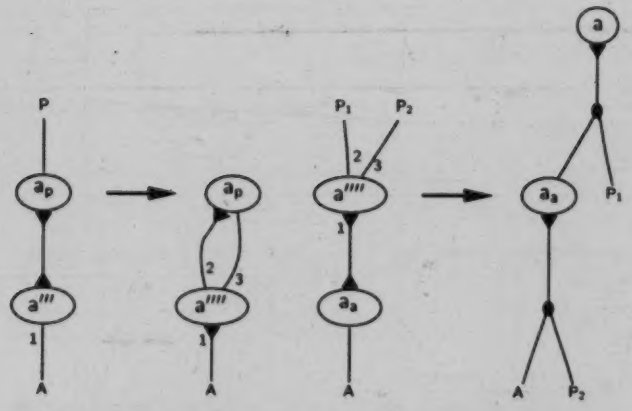
Next the responses of the two hosts interact with each other. Basically, we implement a logical AND: if either host sent a rej, the pre-commitment will be aborted. [⋈ 4.18]



⋈ 4.18: Negotiation: Resolution of Individual Host Responses.

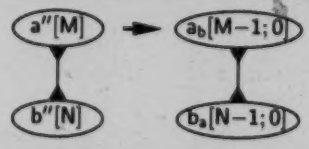
Note that the end of item 4 above states that we should return the port-diamond to its initial state. To this end, we introduce auxiliary states a''' and a'''' which "find" an a_p node on a's principal port and reconnect its ports to the ports of a_a. This may pick *any* a_p port, but that is ok since all of them are mutually replaceable, and anyway the diamonds are shared at the principal port of a_p to begin with. [⋈ 4.19]





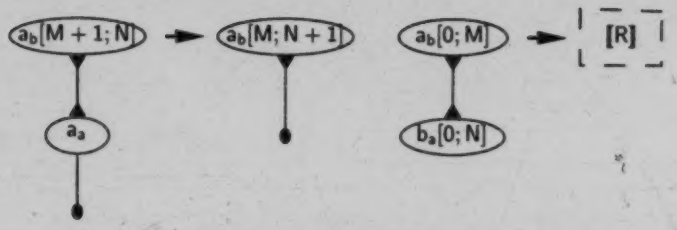
⋈ 4.19: Rejection and Restoration of the Port Diamond.

In case of successful negotiation (two ack nodes produced), both hosts become fully committed by the last rule of ⋈ 4.18 (and its analogue for b). We now proceed with implementing the interaction proper. First a'' and b'' decrement the arities of their principal ports (we indicate only these arities; the number after the semicolon is an extra "register" initialized to 0). [⋈ 4.20]



⋈ 4.20: Start Implementing the Interaction: Decrement the Principal Port Arities.

Then, assuming that the principal port of a is passivated in the RHS of the rule, a_b proceeds to passivate its diamond bundle by removing a_a nodes. Similarly for b_a if b's principal port is passivated. While passivating a_a nodes, a_b "counts down", in order to know when it is done. If the principal port of a is not passivated in the RHS, then ⋈ 4.20 should produce $a_b[0; M-1]$ right away. [⋈ 4.21]



⋈ 4.21: Passivate the Principal Edges of a; Produce RHS.

Finally, a_b and b_a produce the translation of the RHS of the original rule $a \bowtie b \to R$. They only have to connect their free ports to the free ports of $[R]$, and to "pass" the arities M and N to the recipient of their principal port "bundles".

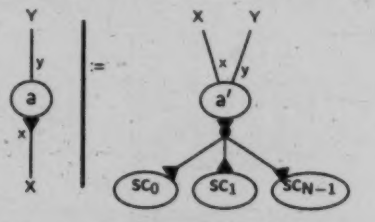
4.5 Representing INMR as INMP: Self-Commitment Nodes

It is easy to emulate INMR as INMP by introducing auxiliary *self-commitment* nodes sc_i that commit an INMR node a to one of the possible choices it can take in the different alternative INMR rules applicable to a given redex, say $a \bowtie b$. Such a commitment may be made at different stages of the evolution of a. Intuitively, a makes the commitment "late", just before $a \bowtie b$. We could use auxiliary nodes on the cut edge a–b itself, in order to model such a late commitment. However, since we won't investigate behavioral correspondence of translations in this section, for simplicity sake we attach the commitment nodes to a itself. The commitment can be made at any stage of the evolution of a.

There are various ways of ensuring that commitment nodes provide "enough" choices so that any of the alternative rules for $a \bowtie b$ can materialize. In order to achieve some fairness, we introduce the following construction. Let $n_{a \bowtie b}$ be the number of rules applicable to the redex $a \bowtie b$ in a given INMR system IN. Let

$$N = \mathrm{lcm}\{n_{a \bowtie b} \mid a, b \in \mathrm{IN} \wedge n_{a \bowtie b} > 0\}$$

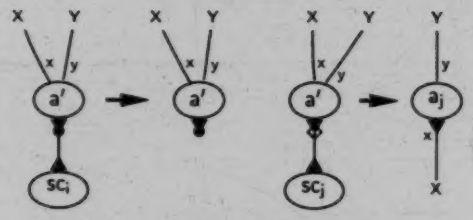
be the least common multiple of all such numbers. Then we translate an INMR node a (shown on the left) to the INMP configuration shown on the right: [⋈ 4.22]



⋈ 4.22: Representing INMR as INMP (Step 1)

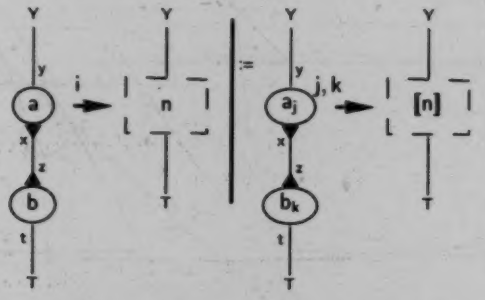
Here we attach N different self-commitment node types sc_i to a, which nodes may determine the "fate" of a in N different ways.

The translated a′ then kills all but one of the self-commitment nodes, which it chooses non-deterministically. Let's assume that the chosen self-commitment is sc_j. Then a′ "confirms" this choice by becoming a *committed alternative* node a_j. [⋈ 4.23]



⋈ 4.23: Representing INMR as INMP (Step 2)

Now let's consider the ith of the $n = n_{a \bowtie b}$ INMR rules applicable to the redex a ⋈ b ($i = 0 \ldots n-1$). We translate the ith rule (given on the left) to the set of rules given on the right, where $j, k = 0 \ldots N-1$ and $j + k = i \pmod n$. [⋈ 4.24]



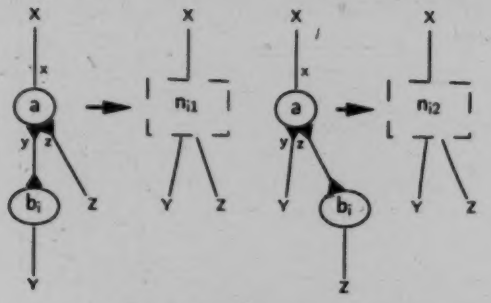
⋈ 4.24: Representing INMR as INMP (Step 3)

The condition $j + k = i \pmod n$, together with the condition that N is divisible by n, ensures that every sum $j + k \pmod n$ has equal frequency of appearance over the domain $j, k = 0 \ldots N-1$. If all sc_j of ⋈ 4.23 have the same "chance of survival", this gives an equal (fair) activation chance of each one of the n alternative rules.

Our translation above is quite wasteful of node types (the same could be achieved with a lot less node types), but it has the advantage of being simple and easier to understand.

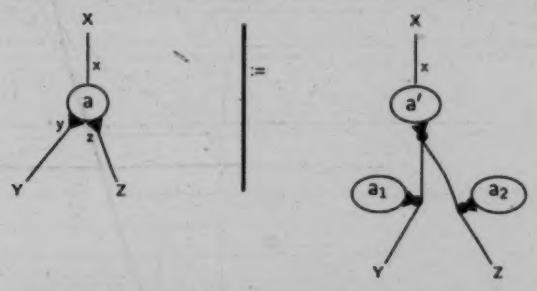
4.6 Representing INMPP as INMP: Marker Nodes

We try to emulate INMPP in INMP by splitting a node with n principal ports into $n + 1$ nodes, each with one principal multiport. For simplicity, we show here the case of $n = 2$. Let a be a node type with two principal ports y and z and let its reductions with different node types b_i be given by the rules [⋈ 4.25]



⋈ 4.25: Representing INMPP as INMP (Step 1).

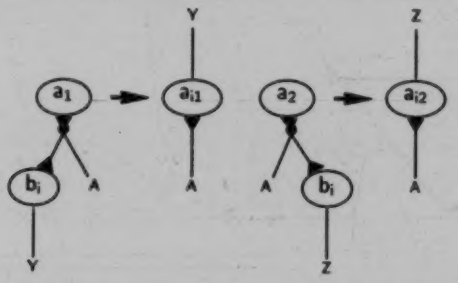
We translate the INMPP node a(X, Y, Z) to the following INMP configuration: [⋈ 4.26]



⋈ 4.26: Representing INMPP as INMP (Step 2).

Here we split a to a *main* representative a′ and n auxiliary *markers* a_j. The main representative a′ has the same ports as a, except that the n principal ports of a are replaced with one principal multiport, which will have arity n (will have n incident edges) at all times. The markers have only one port, a principal multiport which will have arity 2 at all times. Intuitively, the role of the markers is to distinguish the n edges emanating from the main representative, which are otherwise indistinguishable.

The markers are governed by the following rules. First, when a marker a_j meets one of the nodes b_i that can interact with a, an interaction occurs that commits the marker to a transitionary node type a_{ij}.[7] This node represents a *pre-commitment* to the interaction $a \bowtie b_i$. [⋈ 4.27]

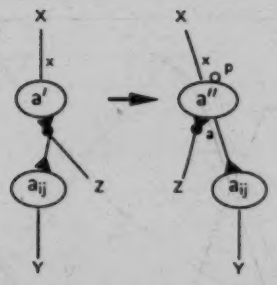


⋈ 4.27: Representing INMPP as INMP (Step 3).

(Nominally the principal port of a_{ij} should be a multiport, so that it can take any left-over edges of a_j. However, as mentioned above, a_j has exactly two edges at all times, so we forego this complication. Of course, in a proof of the faithfulness of the translation, this arity invariant must be proven formally.)

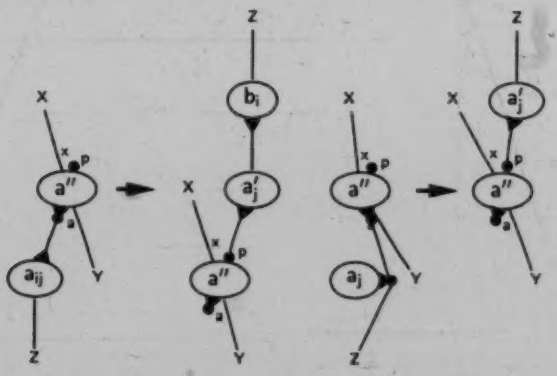
In the next step, the main representative a′ "chooses" non-deterministically one of the pre-commitments a_{ij} and commits to the interaction represented by it. a′ "becomes" a *committed* node a″, which has an *active* single-port a and a *passive*, initially empty, multiport p. The commitment of a″ to a_{ij} is expressed by the attachment of a_{ij} to the active port a. [⋈ 4.28]

[7] Actually things are more complicated than that. If b_i itself had multiple principal ports, it will be split by the translation into multiple nodes too. In that case we need to consider b_{ik} and transitionary node types a_{aj}.



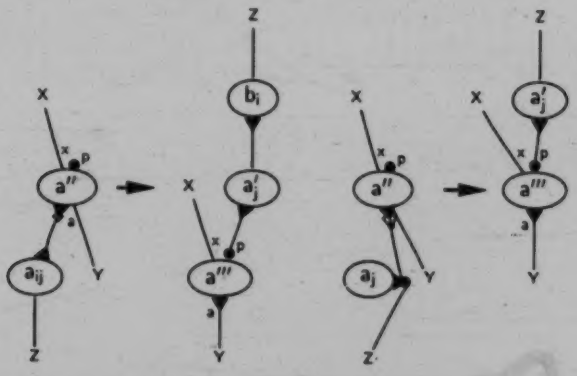
⋈ 4.28: Representing INMPP as INMP (Step 4).

The commitment a″ then proceeds to "passivate" non-chosen pre-commitments ("false firings") a_{ij} by converting them to *passive markers* a'_j and moving them to its passive port p. It also passivates non-chosen markers a_j that have not fired.[8] [⋈ 4.29]



⋈ 4.29: Representing INMPP as INMP (Step 5).

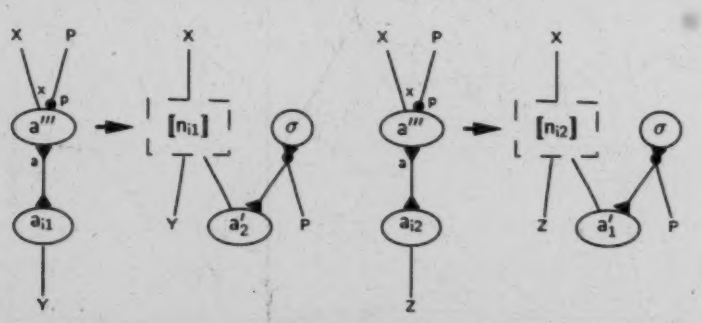
We need to duplicate the above rules and to add arity constraints, in order to introduce a last (finalizing) step for the above iterative passivation process: [⋈ 4.30]



⋈ 4.30: Representing INMPP as INMP (Step 6).

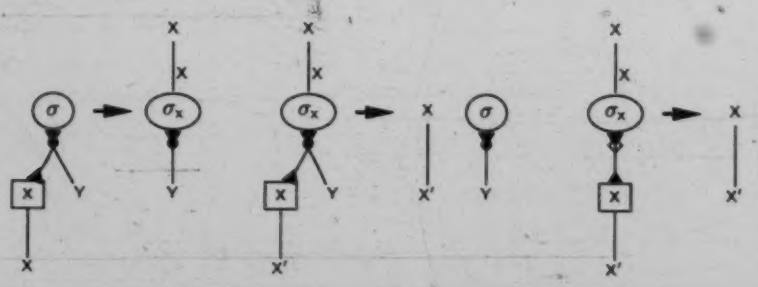
[8]For this latter rule, again nominally the auxiliary port of a'_j should be a multiport in order to accommodate any left-over edges of a_j, and again we use the invariant that the arity of a_j is 2 at all times.

Once the committed node a'' passivates all non-chosen markers and pre-commitments, it becomes a''' and "turns its attention" to the active pre-commitment a_{ij}. This pre-commitment determines the resulting net $[n_{ij}]^9$ to correspond to the rules ⋈ 4.25, while adding some extra nodes a'_j and σ. [⋈ 4.31]



⋈ 4.31: Representing INMPP as INMP (Step 7).

The *short-circuiting* node σ has the important task of reconnecting the free edges of $[n_{ij}]$ to their counterparts waiting on the p port. σ accomplishes this by short-circuiting in pairs same-named nodes located at its port, and then disappears. Here the "abstract" node x may represent any node type. [⋈ 4.32]



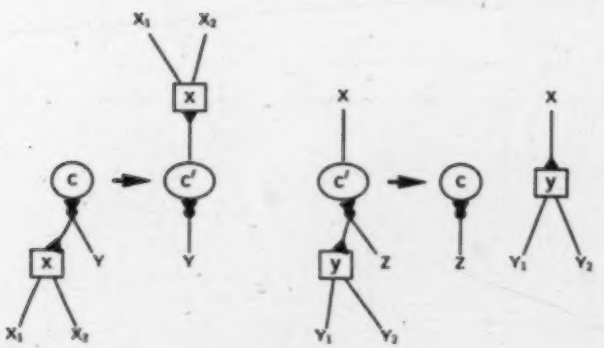
⋈ 4.32: Representing INMPP as INMP (Step 8).

The rules ⋈ 4.32 depend on the invariant that the nodes connected to σ can be partitioned into a number of pairs of nodes of the same type. This invariant can be proven easily by inspecting rules ⋈ 4.29 and ⋈ 4.30 for what gets attached to the port p, and rule ⋈ 4.31 for how this edge bundle is migrated to σ, and a last node a'_j is added.

4.7 Representing INMC as INMP: Explicit Connector Nodes

We will attempt to emulate INMC in INMP by introducing an explicit *connector* node c with a single principal multiport. It picks at random two ready-to-interact nodes x and y amongst the ones connected to it, and puts them in contact. [⋈ 4.33]

[9] Here $[\cdot] : \text{INMPP} \to \text{INMP}$ is the translation function that we are defining in this section.



⋈ 4.33: Connector Node c.

There are several problems with this naive approach:

Premature commitment The first rule above commits to an interaction involving x too early. The connector has not checked that there is an appropriate node y ready to interact with x at this moment. At the same time, there may be another pair of nodes attached to the same connector (say x_1 and y_1) which are ready to interact. In this case the premature commitment will block the interaction $x_1 \bowtie y_1$, since c' looks for neither x_1 nor y_1.

Connected connectors Two connection points may become conjoined in an INMC for several different reasons. This should be modeled by the merging of two connectors, but the rule above does not make provisions for such merging. According to it, only one edge of the first connector will be migrated to the second, while we need to make possible the interactions implied by all edges.

We tackle these issues in the subsections that follow.

4.7.1 Avoid Premature Commitment Using Counting

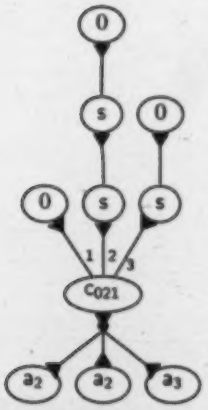
In order to prevent blocking through premature commitment, we will use more "intelligent" connector types, which will know at all times how many nodes of each node type are connected to it. (In more technical terms, know the cardinalities of the multiset of nodes connected to it.) Let $a_1 \ldots a_n$ be all the node types of the original INMC system that we are emulating.

We implement the counting machinery in two parts:

1. We use n auxiliary ports, the ith of which keeps the number of a_i nodes hosted at the connector, using an unary-arithmetic chain (⋈ ??).

2. We use 3^n connector types $c_{b_1 \ldots b_n}$ where $b_i \in \{0, 1, 2\}$ indicates whether there is 0, 1, or more a_i nodes at the connector.

For example, if $n = 3$, then a connector hosting [0,2,1] nodes of types a_1, a_2, a_3 respectively will look like this: [⋈ 4.34]

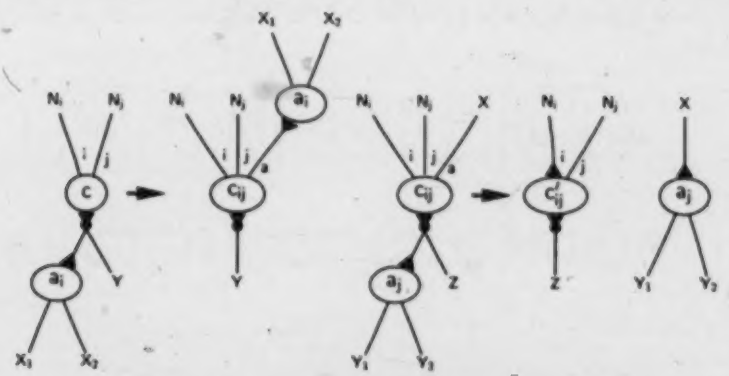


⋈ 4.34: Counting Connector.

The reason why we need the second ("internalized") part of the counting machinery is so that the connector can make right choices about commitment atomically, and won't have to backtrack (undo) commitments. The reason why we need to keep ternary digits b_i and not simply bits[10] is because the original system may allow reflexive interactions $a_i \bowtie a_i$, in which case we need to know that there are no less than 2 a_i's available at the connector before we can commit to such interaction. We hardwire in the rules governing connectors the knowledge of which node types can interact with which other node types.

Irreflexive Interactions

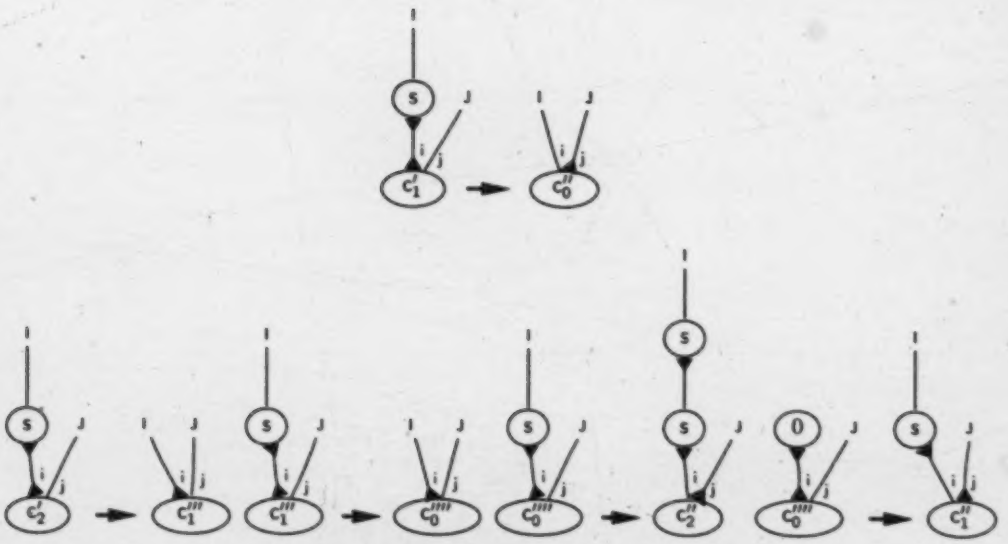
For all interactions $a_i \bowtie a_j$ $(i \neq j)$ that the original system permits, we introduce committing rules similar to ⋈ 4.33 for those connector types $c_{b_1 \ldots b_n}$ for which $b_i \geq 1 \wedge b_j \geq 1$. (Such connector types are indicated simply with c below, and for simplicity we've omitted all counting ports but i and j.) [⋈ 4.35]



⋈ 4.35: Connector Commitment for Irreflexive Rules.

Now the transitional connector state c'_{ij} must decrement the unary numbers at ports i and j, and adjust the bits b_i and b_j. It does this in several transitional steps, where the first group of steps handles i and b_i. For simplicity of notation we now omit the indexes i and j and the principal multiport of the connector, and use a subscript for the bit b_i, *i.e.* write c'_{b_i}. [⋈ 4.36]

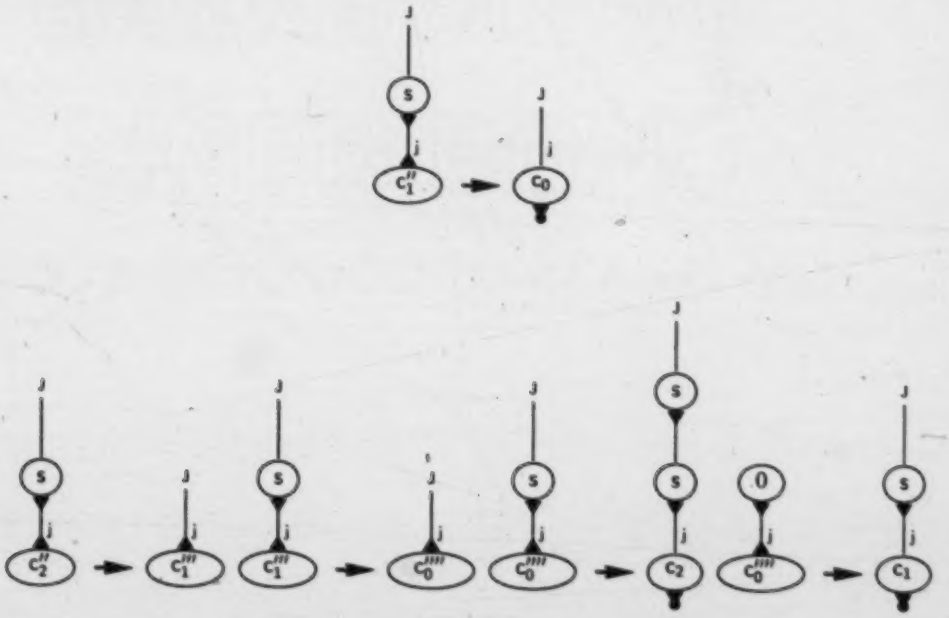
[10] However we still call them "bits" in what follows, for the purpose of more fluency in writing. We prefer this slight abuse of the established term "bit", rather than introducing a new and ugly term like "trit" or "ternit".



⋈ 4.36: Decrementing Connector Counter i.

The first rule $c_1' \to c_0''$ simply decreases the number at port i and the bit b_i. The second rule (c_2') also decreases the number at port i, but can decrease the bit b_i only *tentatively*, because the value $b_i = 2$ means "2 or more". Therefore we check two elements down the unary chain at i, and the intermediate state c_0'''' becomes either c_1'' or c_2'', depending on whether there were only 1 or "2 or more" elements in that chain.

In the second group of steps, the node c_{b_i}'' proceeds to decrement the unary number at j and the corresponding bit b_j. Again, for simplicity of notation we omit b_i, and use a subscript for the state of b_j. [⋈ 4.37]

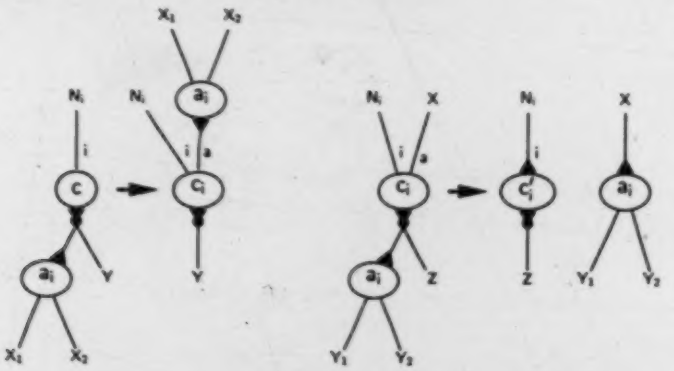


⋈ 4.37: Decrementing Connector Counter j.

The first rule (c_1'') decreases the number at j and the corresponding bit b_j. The second rule (c_2'') can decrease the bit b_j only *tentatively*, because it means "2 or more". Therefore the latter rules check two elements down the unary chain at j. The intermediate state c_0''' becomes either c_1 or c_2, depending on whether there were only 1 or "2 or more" elements in that chain. In all cases, at this point the connector returns to its main type c.

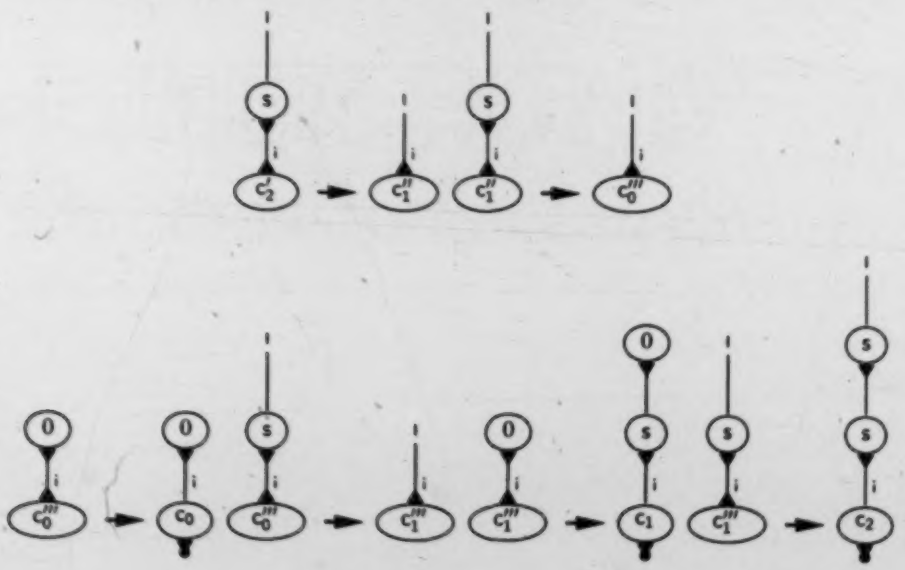
Reflexive Interactions

For all reflexive interactions $a_i \bowtie a_i$ of the original system, we introduce committing rules for those connector types $c_{b_1 \ldots b_n}$ for which $b_i = 2$. (We indicate below such connector types simply with c_i and for simplicity omit all counting ports but i.) [⋈ 4.38]



⋈ 4.38: Connector Commitment for Reflexive Rules.

Now the transitional connector state c_i' must decrement the unary number at i by 2, and adjust the bits b_i. For simplicity of notation we omit the index i and the principal multiport of the connector, and use a subscript for the bit b_i, *i.e.* write c_{b_i}'. [⋈ 4.39]



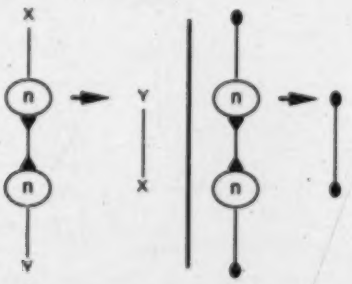
⋈ 4.39: Decrementing Connector Counter.

The rules $c_2' \to c_1'' \to c_0'''$ decrement the unary number at i. At this point we must check what is the new number at i, in order to determine the new value of the bit b_i. After that, the connector returns to its main type c.

4.7.2 Merge Connected Connectors

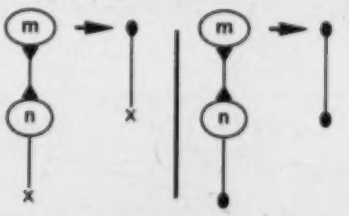
Two INMC connection points may become conjoined (come in contact) in at least two different ways:

- The RHS of a rule has a *shortcut edge*, and the rule is applied to a net where two connection points occur at the LHS interface points corresponding to the ends of this edge. For example, given the rule on the left of the figure below, the reduction shown on the right leads to conjoined connection points. [⋈ 4.40]



⋈ 4.40: Merging of Connection Points Through a Shortcut Edge.

- The RHS *introduces* a connection point that is also connected to the interface of the RHS, and another connection point happens to occur at that interface. Here is an example: [⋈ 4.41]



⋈ 4.41: Merging of a Newly-Introduced Connection Point with an Existing One.

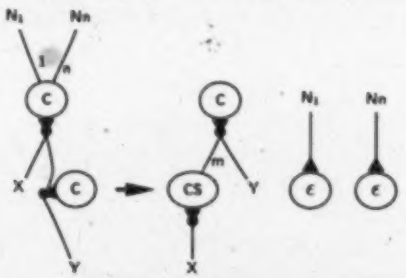
In both of these cases, our translation to INMP must *merge* the two connectors representing the connection points: migrate all edges from one to the other, migrate the counters, then delete the empty connector.

There is one problem with this description: the two connectors may be exactly the same (have the same bit-counts), so it is not clear how to determine the direction in which the edges and counts (the "assets"of a connector) should migrate. We can define some unoriented asset migration rules, but then we take the risk of divergent computations: an asset moving back and forth between the two connectors, and the migration not making any progress.

Another approach is to introduce somehow a distinction between the two connectors, so that one takes the role of source and the other takes the role of target. There doesn't seem to be any "natural" distinction that we could use; we could try to make newly-created connectors targets so that they "attract and consume" existing connectors, but since IN reduction is completely local, it

is possible that two such new connectors may come in contact as the result of two "adjacent" rule reductions.

Therefore, we introduce the following simple commitment rule. (We need one instance of the rule for every pair of connector node types $c_{b_1 \ldots b_n}$ and $c_{b'_1 \ldots b'_n}$.) [⋈ 4.42]

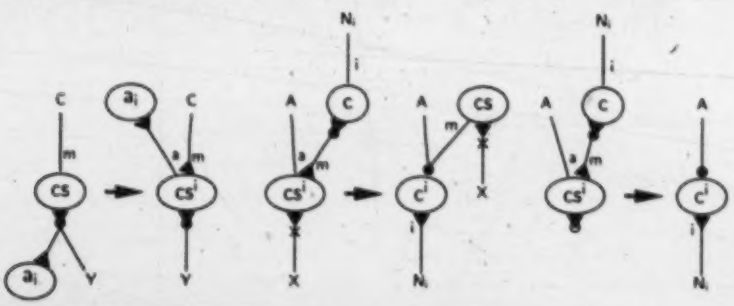


⋈ 4.42: Connector Commitment Produces a *Connector Source* cs.

Please note that this rule is not an INMP rule, since the LHS is symmetric,[11] but the RHS is asymmetric (indeed, that is the whole point of the rule). IN and INMP rules insist that if the LHS is symmetric then the RHS should also be symmetric, so that no matter which way the LHS is matched to an occurrence of it in a net, there is only one possible outcome of the rule's reduction. The rule above is an INMR rule, since INMR allows more than one rule per redex. It seems that we have to accept this bit of "INMR impurity" in our otherwise purely-INMP system.

The *connector source* cs has only one "purpose in life": to migrate its edges to its *master* c, then disappear.[12] We won't introduce rules by which cs can facilitate the interaction of a_i nodes hosted by it[13] (see §4.7.1 and §4.7.1). If two nodes a_i and a_j hosted by a cs "want" to interact, they'd first have to move to cs's master c. Therefore, we don't need the count sequences of cs either, and that is why ⋈ 4.42 erases them. The master will count the assets of the slave anew, as they are being transferred to the master.

Now we go on to define the connector merging (edge migration) process. Due to the INMP restriction that multiport edges may only be migrated one by one (§3.3.1, item 3), and since interaction only occurs on principal ports, an a_i edge cannot be migrated until a_i presents its principal port to cs. Thus migration occurs gradually, as a_i nodes "become activated" one by one. While cs only migrates edges towards the master c, c itself not only accepts them, but may also facilitate interactions between a_i nodes hosted by it. Thus edge migration and connector-hosted interaction are interleaved. If we don't allow such interlacing then connector merging may dead-lock, with cs waiting for the activation of an a_i, which activation can only occur through a c-facilitated interaction. We define the following rules for all $i = 1 \ldots n$ (for simplicity, we leave out all count ports except i). These rules migrate one a_i edge. [⋈ 4.43]

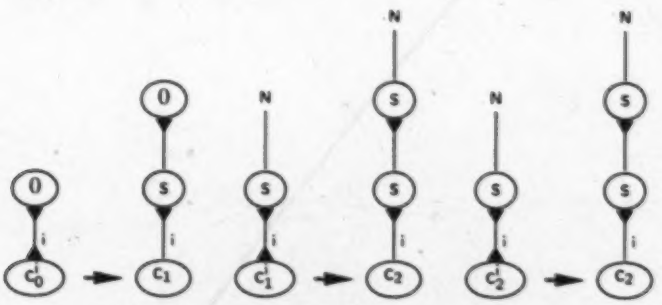


[11] More precisely, for some instantiations of $b_1 \ldots b_n$ the LHS is symmetric.
[12] Thus another good reading of the name cs is *connector slave*.
[13] Here $a_1 \ldots a_n$ are all the node types of the original system, as defined in the beginning of §4.7.1.

⋈ 4.43: Connector Merging: Migrate a_i Edge.

The third rule removes cs if the migrated edge was its very last edge (cs' doesn't have any edges on its multiport).

The following rules increase the ith count sequence of c, and update the corresponding bit b_i. (The subscript of c indicates the value of the bit, and for clarity we have omitted all other count ports and bits.) [⋈ 4.44]

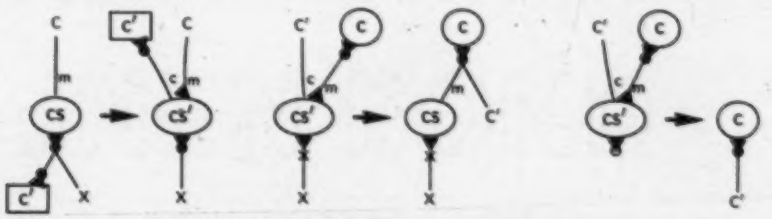


⋈ 4.44: Connector Merging: Increase Count.

4.7.3 Complex Connector Arrangements

The rules of §4.7.2 migrate a_i edges from cs to c. However, there may be other edges attached to a cs besides a_i edges. Namely, there may be other connectors c or connector sources cs attached to it. This may happen since in the completely-local IN framework we cannot enforce the prompt execution of connector merge rules at the expense of all other rules, and new connectors may be created as the result of the execution of such other rules.

In the rules below, the "abstract" (boxed) node type c′ may be any one of c or cs. The rules migrate one connector edge from cs to its master, and if this was the last edge of cs, the last rule also removes cs. [⋈ 4.45]



⋈ 4.45: Migration of Connector Edges.

Note that in the case when $c' = cs$, the first rule is again not an INMP rule but an INMR rule, because the LHS is symmetric while the RHS is not.

To summarize the rules of the previous and this subsections:

1. Connector commitment ⋈ 4.42 turns c into a connector source (slave) cs, which does not facilitate interaction of any nodes hosted by it, rather only migrates edges to its master connected to the cs.m port.
2. Edge migration includes not only "object" edges a_i ⋈ 4.43, but also "subject" edges c−cs ⋈ 4.45.
3. When all its edges are migrated, cs is removed.

Let's consider the connected components (equivalence classes under the reflexive transitive closure of the set of all edges) of an INMC-INMP net, restricting our attention to connectors c and cs only. Once a connector joins such a component C, it should not leave it until the eventual death of the connector. This is required by our intended use of connectors to model INMC connection points, and the semantics of two such points becoming joined.

In order to ensure that our translation works, we must prove the following:

Proposition 4.7.3.1 *Let e_1 and e_2 be two edges which at a certain time become connected to the same component* C. *Then:*

1. *e_1 and e_2 will remain connected to the same component throughout their lifetime.*
2. *If at a certain time e_1 and e_2 are "active" (their other ends are at principal ports), then there is a reduction sequence consisting of migration of* c *and* cs *edges that puts e_1 and e_2 in contact.*

Proof

1. This is easy to see, since a component never breaks apart: cs migrates edges to its master, who is a member of its component, and disappears only when it doesn't have any more edges.
2. Let c_1 and c_2 be the two connectors (or connector sources) hosting e_1 and e_2 respectively. We prove item 2 by induction on the length of a shortest path of connector edges between c_1 and c_2 (there is one since they both are in C). If we can "collapse" every edge along that path, then we can "collapse" the whole path. We prove the base case (one edge between c_1 and c_2) by case analysis on the possibilities for c_1–c_2:
   (a) $c_1 = c \bowtie c = c_2$: Turn c_2 into a cs by $\bowtie$ 4.42 and then migrate e_2 to c_1 by $\bowtie$ 4.43 and $\bowtie$ 4.45.
   (b) $c_1 = c \bowtie cs.m = c_2$: Migrate e_2 to c_1 by $\bowtie$ 4.43 and $\bowtie$ 4.45.
   (c) $c_1 = cs.m \bowtie cs.m = c_2$: This case is impossible, since it cannot occur as a result of our rules.
   (d) $c_1 = c$–$cs = c_2$: Using corollary 4.7.3.3, there exist a unique "ultimate master" of c_2, call it c_2'. Then c_2' is a c node and is connected to c_2 by a oriented path of cs.m–cs edges. Migrate both edges c_1–c_2 and e_2 along this path to c_2'. This puts c_1 and c_2' in contact, which leads us back to case 2a.
   (e) $c_1 = cs.m$–$cs = c_2$: Similarly to the previous case, "replace" c_2 with its ultimate master c_2', which leads us back to (the mirror image of) case 2b.
   (f) $c_1 = cs$–$cs = c_2$: Similarly "replace" both c_1 and c_2 with their ultimate masters c_1' and c_2', which leads us back to case 2a.

   The other possibilities for c_1–c_2 are mirror images of ones above (with c_1 and c_2 swapped), and are proved similarly. ◊

Note that the connected components of a net are dynamic (interaction rules can add connectors at any time). The net is not a static graph that is reduced only by connector rules. However, the dynamicity is limited to only introducing further connectors, so the components only increase as a result of that, which is the essence of 1.

As for 2, the introduction of new connectors can easily increase the path between c_1 and c_2. However, 2 only claims that there exists a reduction that brings c_1 and c_2 together, not that every possible reduction will join them in a finite amount of time.

Lemma 4.7.3.2 (No slave-master cycles) *The rules that we use in the translation of INMC to INMP (§4.7) don't allow the construction of an oriented cycle consisting of edges heading from* cs.m *ports. (We denote such edges as* cs.m–*c where c can be either* c *or* cs.*)*

Proof cs.m−c edges are introduced only by rule ⋈ 4.42, which creates a cs.m−c edge. Later the tail c may be replaced with a cs node by another application of the same rule to c and another c node. There is no way for the path to loop back onto itself, since all its nodes but the last one are cs nodes. ◊

Corollary 4.7.3.3 (Ultimate master) *The oriented path of* cs.m−c *edges starting at a given* cs *node always leads to a unique* c *node, which we call the* ultimate master *of* cs.

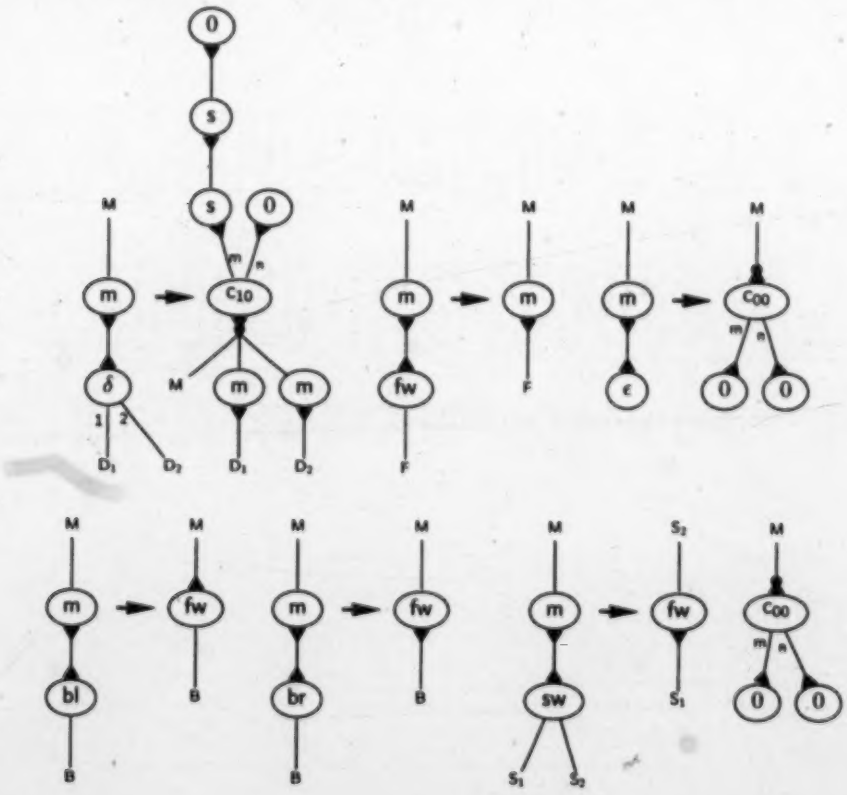
Proof Use the previous lemma and the fact that INMPs are finite nets. ◊

4.7.4 Optimize the Translation Using Typing (cc-INMP Example)

If some kind of node type "abstraction" is available in the original INMC system, we can use it to minimize the amount of counting ports, bits, and therefore rules. In IN, often a "type discipline" is used to "collapse" (abstract away) node types, and to make the applicability of rules more uniform and simpler to determine.

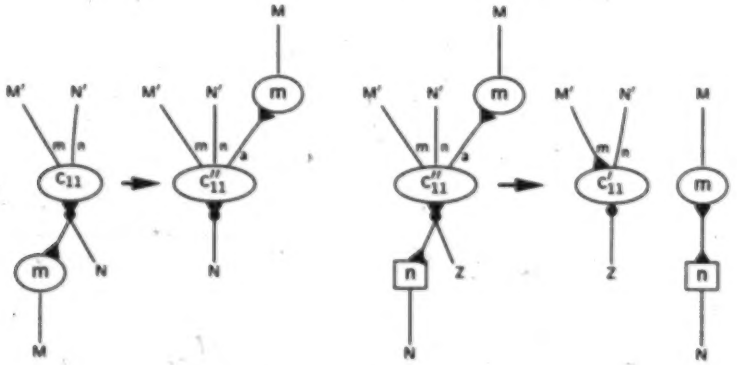
For example, in *cc* process graphs (§3.4.1), the *message* node m is allowed to interact with any other node type, and no other interactions are allowed. Therefore, a connector needs to keep only two counters m (for m nodes) and n (for all other node types). Since there are no reflexive rules, the connector can be limited to maintain simple 2-valued bits instead of 3-valued ternary digits.

This simplifies the translation to only four connector types $c_{b_m b_n}$, some auxiliary (temporary) types, and the original *cc* types. In a first step, we translate the INMC system for *cc* ⋈ 3.29 to the following INMP system (bold dots representing INMC connectors are replaced with explicit INMP connectors): [⋈ 4.46]



⋈ 4.46: Representing cc Graphs as INMP (Step 1).

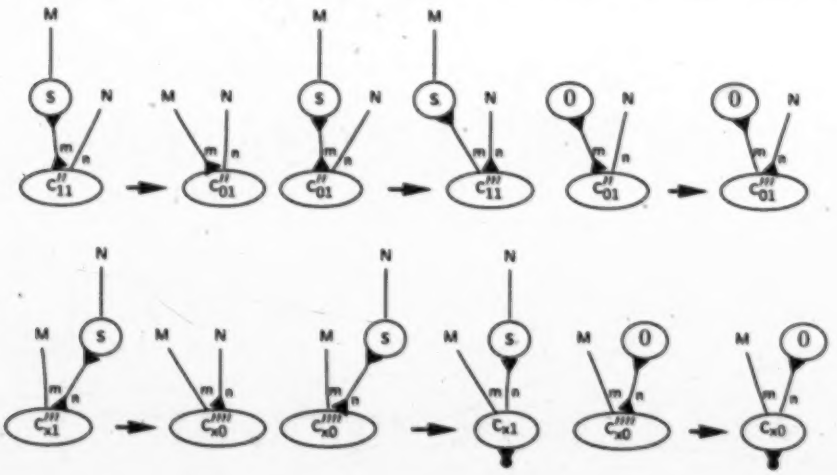
The connector rules include the normal connector merging rules of §4.7.2. They also include a specialization of irreflexive commitment ⋈ 4.35, shown below: [⋈ 4.47]



⋈ 4.47: Representing cc Graphs as INMP (Step 2).

Above n indicates any non-m node. We have determined arbitrarily that c_{11} will first seek an m node (this has no effect because it seeks a n node in the very next step).

They also include counter decrementing rules similar to ⋈ 4.36 and ⋈ 4.37, namely: [⋈ 4.48]

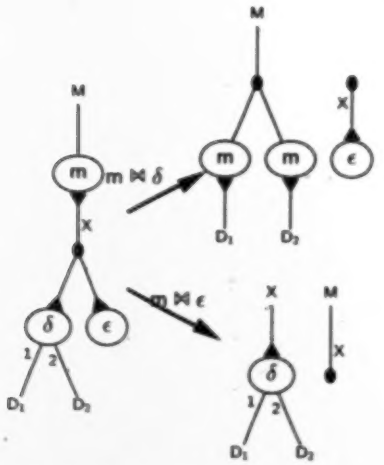


⋈ 4.48: Representing cc Graphs as INMP (Step 3).

Here c_{x1} indicates either c_{01} or c_{11}, and analogously for c_{x0} and for the primed versions.

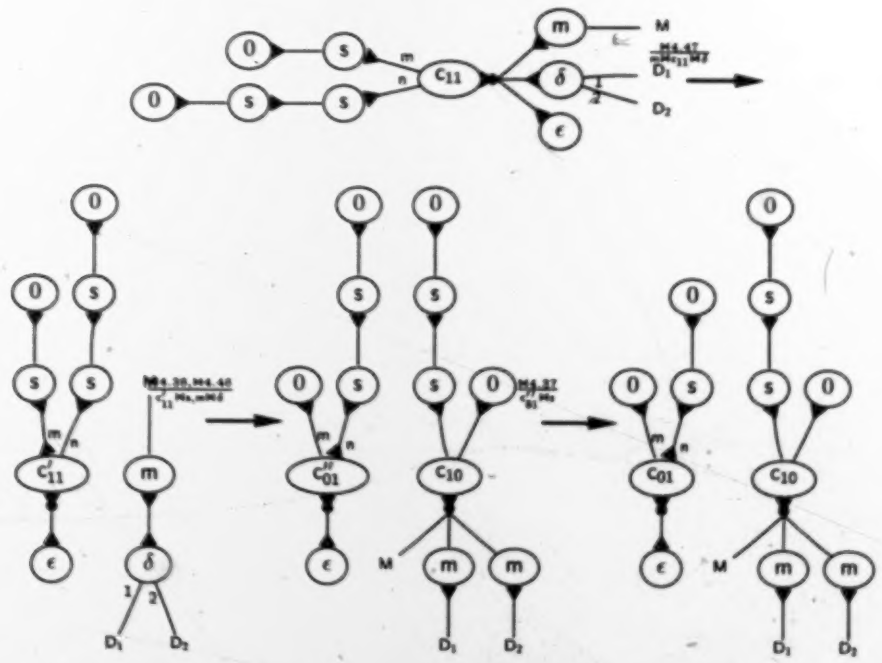
Finally, we need connector commitment ⋈ 4.42 and connector merge rules, the same as in §4.7.2 and §4.7.3.

Example of cc-INMP Reduction Now we give an example of reduction in the INMP system introduced above (let's call it cc-INMP). We will translate the example of Figure 9 in Yoshida (1994) and trace the two possible reductions. [⋈ 4.49]



⋈ 4.49: Sample *cc*-INMC Graph.

We translate the initial configuration above to the following cc-INMP net, and start emulating the upper reduction. Here the complex label ("ratio") over the reduction arrow indicates (above the line) the rule(s) that are applied, and (below the line) the node types of the nodes that interact. [⋈ 4.50]

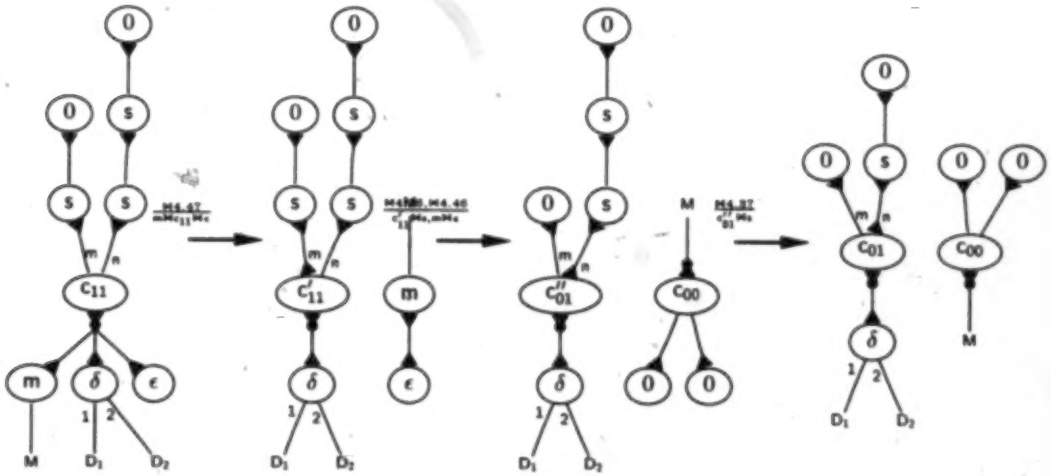


⋈ 4.50: Corresponding cc-INMP net; Emulating the Upper cc-INMC Reduction.

The connectors in the final configuration accurately reflect the counts of nodes hosted at them: [0,1] on the c_{01} connector and [2,0] on the c_{10} connector (where the numbers $[m, n]$ give the number of m and non-m nodes respectively). We also see that in order to read the result back (reinterpret it

in INMC), we need a simple transformation to convert connectors back to INMC connection points. (In more complex cases, when there are several connected connectors in the result, we'll need to apply the collapsing rules of §4.7.3.)

We now emulate the lower reduction of ⋈ 4.49 as follows: [⋈ 4.51]



⋈ 4.51: Emulating the Lower cc-INMC Reduction.

4.8 Future Work

Future work should address the following topics:

- Formal proofs of representability. This would involve the development of appropriate behavioral relations, for example variants of bisimulation.
- Well-behavedness of representations (convergence, determinism).
- Complexity (cost) of representation, along the following lines:
  - Number of node types in the target system, in relation to the number of types in the source system.
  - Number of ports per node in the target system.
  - Number of target nodes in the representation of a single source node.
  - Number of target reductions per source reduction; fluctuation of this number in various stages of target reduction.

Chapter 5

Representing the Finite π-calculus in Multi-Interaction Nets

Concurrency = Interaction + Non-determinism

In this chapter, as an application of the non-deterministic mechanisms to IN that we introduced earlier, we study a representation of the finite monadic π-calculus in a non-deterministic IN system. Similarly to the λ-calculus, the π-calculus leaves some of the basic low-level components of computation implicit, namely the distribution of values and synchronization, expressed as the global process of substitution. Uncovering this finer computational structure, and dispensing with the important role that syntactic entities like names play in the π-calculus, is the goal of this chapter.

For convenience, in this chapter we use a system that is a combination of INMP and INMPP: both multiple principal ports and multiports are allowed. We call this system *Multi-Interaction Nets*, or MIN. Please note that a simple adaptation of our translation of INMPP into INMP (§4.6) would allow us to reduce MIN to INMP, and thus to migrate the results of this chapter from the richer system MIN to the simpler system INMP.

5.1 The π-calculus

The π-calculus of Milner *et al.* (1992) is one of the most popular theoretical tools for the investigation of concurrent computations. Its popularity is due to its conceptual simplicity, yet great expressive power.

In this chapter we consider the finite part[1] of the monadic π-calculus. The π-calculus is a *process calculus*, a formalism that allows us to study formally systems of interacting processes.

In the rest of this section we define the π-calculus formally, and then discuss some properties of the *prefix operator*.

[1] The *finite* π-calculus does not include the operators of *choice* (alternative between two processes) and *replication* (the re-establishment of a process for future reuse). These operators are analogous to if and while of procedural programming languages, and are crucial for the functional completeness of the π-calculus. However, they are not critical for understanding what we consider the essence of the π-calculus, namely the passing of channel names as values, and thus dynamically reconfigurable communication structures. We discuss how we could accommodate choice and replication in §5.8. Please note that *e.g.* Honda and Yoshida (1994a) take a similar approach, and later develop their combinatory system to accommodate these elements in (Honda and Yoshida, 1994b).

5.1.1 Definition of π-calculus Processes

Let $a, b, \ldots, x, y, \ldots$ and their subscripted and primed variants stand for *channel names*.[2] Then *processes* $P, Q, \ldots$ are defined inductively as follows

Zero 0 is the empty (do-nothing, inactive) process.

Parallel Composition P, Q behaves as both P and Q, possibly interacting with each other.

Output Prefix $c!v.P$[3] sends the value v along the channel c, then behaves as the process P.

Input Prefix $c?x.P$ receives a value (say v) from the channel c, then behaves as the process P, where the value v is substituted for the free occurrences of the name x in P. The name x is bound by this prefix.

Hiding/Restriction $(c)P$ is just like P, but the channel c is hidden (bound): no values can be sent/received over that channel by other processes. Therefore, c becomes a private channel of P, it cannot be "seen" on the outside of P, and the outside cannot communicate with P on that channel, nor can it interfere with P's private communications on that channel.

We will sometimes denote a prefix $c!v$ or $c?v$ as π. We abbreviate $\pi.0$ to simply π (an input/output *atom*). In the prefixes $c!v$ and $c?v$, the name c is said to be in *subject* (channel, active) position, while the name v is said to be in *object* (value, passive) position. Free and bound names of a process are defined as usual: x is bound in $c?x.P$ and $(x)P$, all other name occurrences are free. $a!(x).P$ is used as an abbreviation for $(x)a!x.P$, which is the output of a bound name.

The most notable feature of the π-calculus compared to earlier "static" process calculi such as Communicating Sequential Processes (CSP) of Hoare (1985) and Calculus of Communicating Systems of Milner (1980) is that channel names can be passed around as values. This allows *dynamically reconfigurable* communication structures (networks), because if there is a channel c between two processes P and Q, then they can pass to each other any number of other channels, channels which create arbitrary communication structures "inside" P and Q. This passing of channels can "connect" the two structures in P and Q and create a bigger "unified" structure. Therefore, in a sense the π-calculus is about switching (wiring, connecting) communication structures dynamically. The π-calculus is the forefather of a growing family of *mobile* process calculi.

5.1.2 Structural Congruence

A natural way to simplify the presentation of the reduction system of the π-calculus (see below) is to introduce structural rules which describe equivalence classes of process terms that are essentially the same, and only differ in their syntactical presentation. $\equiv$ is the smallest congruence satisfying the rules below:

1. $P \equiv P[y/x]$[4] if x is not free in P and y is fresh (α renaming). We will tacitly assume for the rest of this chapter that every bound name is different (renamed apart) from every free name and from all other bound names.
2. $0, P \equiv P \quad P, Q \equiv Q, P \quad (P, Q), R \equiv P, (Q, R)$: the parallel constructor and 0 form a commutative monoid.
3. $0 \equiv (x)0$: there is nothing to hide in 0.

[2] We usually use $a, b, \ldots$ to refer to channel names in their role of *channels*, and $x, y, \ldots$ to refer to channel names in their role of *values*. However, this distinction is merely notational.

[3] We use this Occam-like notation instead of the conventional notation $\bar{c}\langle x\rangle.P$, mainly because it is more clear in the typescript.

[4] Here $[y/x]$ denotes the substitution of y for x.

4. $(x)P, Q \equiv (x)(P, Q)$ if x does not occur in Q: the scope of a restriction can be extended if there's nothing to hide. This also implies that $Q \equiv (x)Q$ given the same proviso. (Given the assumption in 1 that bound names are renamed-apart, the proviso is always satisfied.)

5. $(x)(y)P \equiv (y)(x)P$.

By the last three rules and the assumption that bound names are renamed apart, we can assume without loss of generality that all restrictions are pushed out to the top level.

5.1.3 Reduction Rules

We use an (unlabeled) reduction system for the π-calculus, because we find it more natural for our correctness proofs (later in this chapter) than a labeled transition system. It is the smallest relation $\rightarrow$ closed under the rules

Comm $a?x.P, a!c.Q \rightarrow P[c/x], Q$: this rule, analogous to β-reduction, is the essence of the π-calculus. It allows a process to communicate a name to another process.

Struc If $P \rightarrow Q$ and $P \equiv P'$, $Q \equiv Q'$ then $P' \rightarrow Q'$. This incorporates the structural congruence into the reduction relation.

Par If $P \rightarrow Q$ then $P, R \rightarrow Q, R$: adding a parallel component does not decrease the possibilities for reduction.

Res If $P \rightarrow Q$ then $(x)P \rightarrow (x)Q$: restriction does not decrease the possibilities for reduction of the internal process.

Multi-step reduction is defined as

$$\Rightarrow \stackrel{\text{def}}{=} \stackrel{*}{\rightarrow} \cup \equiv.$$

where $\stackrel{*}{\rightarrow}$ is the iteration (Kleene star) of single-step reduction.

As an example, we give below a simple formalization of the following sequence of communication events: a person talks on the telephone t to another person and receives an email address e. Then s/he sends a message m to that address e, which is received by a third person:

$$t?x.x!m, t!e, e?y \rightarrow e!m, e?y \rightarrow 0, \tag{5.1}$$

and as a "side effect" the substitutions $[e/x]$ and $[m/y]$ remain, and may "act" on the context of the above communication, if x and y appear in that context.

5.1.4 The Many Roles of Prefix

The **Comm** rule plays several important roles that we would like to implement in MIN as independently as possible:

Value passing The value c is delivered from the sender to the receiver process.

Value distribution If the input variable x appears in several places in the receiver P, the value c should be delivered to all these points.

Synchronization Reduction is not a congruence over prefix: $P \rightarrow Q$ does not imply $\pi.P \rightarrow \pi.Q$. In fact, no prefixed process ever reduces on its own, it can only reduce if it meets a process having a complementary prefix/atom. Formally, for every process P we have $\pi.P \not\rightarrow$, where $P \not\rightarrow$ means $\forall Q.P \not\rightarrow Q$. Thus a prefixed process cannot perform any other action before and unless the outer communication is completed.

The prefix blocks any possible interactions inside the prefixed process, as well as interactions between it and other processes. Synchronization in the π-calculus is *global* with respect to the prefixed process. We will see below that our translation to MIN implements synchronization in a distributed manner.

5.2 Representing the π-calculus in MIN

Our motivation for attempting to represent the π-calculus in MIN is threefold:

- To test the expressive power of this extension of IN.
- To represent explicitly some aspects of the dynamics of computation of the π-calculus that are left implicit by the conventional formulation. Namely, the operation of ***substitution*** in the **Comm** rule in §5.1.3 is ***global*** with respect to the receiving process P. In a tradition started by Linear Logic, we would like to represent this operation in discrete local steps.
- To capture the mobility features of the π-calculus in a finitary (name-free, combinatory) framework.

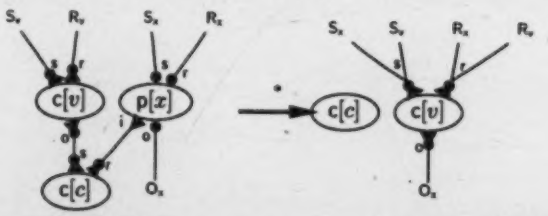
Please see §4.1 for some background material to implementation translations (that section is about translations between various IN systems, but the same ideas apply to a $\Pi \rightarrow$ MIN translation). The basic idea of our translation is to represent every π-calculus channel name as a separate MIN ***node***, and to represent the subject-object relation between the two names in a prefix $c \dagger x.P$ or atom $c \dagger x$ as an ***edge***.[56] Since a channel can participate in several subject-object relations (both as subject and as object), we need ***multiports***. A channel node c needs (at least) four multiports, to capture the possibility of it participating in input/output, and as object/subject. We call these ports input, output, receive and send (i, o, r, s):

Process	Port of c	Net
$x?c$	input	x.r–c.i
$x!c$	output	x.s–c.o
$c?x$	receive	c.r–x.i
$c!x$	send	c.s–x.o

Table 5.1: MIN_π Port Names.

We need to use two different node types (see §5.2.1): c for a ***proper channel***, and p for a ***placeholder***, which is a channel name x in input object position, such as $c?x$. The reason for this distinction is that a placeholder is passive, in the sense that it cannot participate in any communication until it is ***instantiated*** with a name v in a send-receive interaction on its subject channel c, an interaction such as $c?x, c!v$. This instantiation involves the migration of all links (communication capabilities) of x to v.

The essence of our translation is that passing of channel names as values is modeled by "passing" of edges between nodes. Below, c[v] denotes a node of type c bearing a label v (see §5.2.2), and similarly for the other nodes. Very roughly: [⋈ 5.1]



[5]Note that for example, the representation in (Honda and Yoshida, 1994a) is *dual* to ours: a prefix $c!v$ or $c?x$ is represented as a single node. Very roughly, graph edges in that representation correspond to nodes in our representation, and vice versa. We discuss the relation in more detail in §5.7.2.

[6]The edge may be "broken" (made indirect) if the prefix/atom $c \dagger x$ is subjugated to another prefix, see §5.3.

⋈ 5.1: The Basic Idea of the $\Pi \to$ MIN Representation: Translating the Reduction $c?x, c!v \to [v/x]$.

In the above, where we have drawn single edges for O_x, S_x, R_x, there can be in fact many edges, and they all must be migrated to v in the depicted multi-step reduction.[7] Remember that in order not to clutter our diagrams unnecessarily, we often omit edge-less multiports, as we have done here for c[c].o on the LHS and for all ports of c[c] on the RHS.

We also need to use certain *blocking machinery* consisting mainly of *blockers* b and *unblockers* u. The purpose of this machinery is to passivate the process P in an input prefix $c?x.P$, so that it cannot participate in any interactions until the prefix has been "stripped" by a send-receive interaction.

In the rest of this section, we first introduce formally the basic building blocks for representing the π-calculus, namely the nodes and rules of a MIN instance that we call MIN_π. Then we give a translation $[\![\cdot]\!]$ from Π to MIN_π, and finally prove some correctness results about our translation.

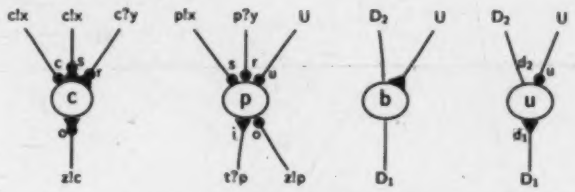
5.2.1 MIN_π Nodes

The node types of MIN_π are classified as two kinds:

primary (main) nodes appear in the translation $[\![P]\!]$ of a process P.

auxiliary (secondary) nodes are used only during intermediate computation steps.

The main node types are: [⋈ 5.2]



⋈ 5.2: MIN_π Main Node Types.

Here we have also shown the port names (*e.g.* i in p.i stands for input), and on the free ports, we have shown the communication *roles* that the nodes can participate in (*e.g.* t?p on p.i means that p is an input object in a prefix or atom $t?p$).

Below are the names of the main nodes, together with a brief description of their function:

Channel node c corresponds to a π-calculus channel name. Its output multiport o connects it to all channels z for which node c is an output object. The send multiport s connects it to all nodes x that are output objects of node c. The receive multiport r connects node c to all nodes y that are input objects of c. The committed multiport c is similar to multiport s, but it holds only output objects x for which we are certain that they are ready to be sent. (The reader is not expected to understand the role of c.c at this point, see ⋈ 5.15 later in this chapter.)

Placeholder node p[8] corresponds to the bound variable in a π-calculus input prefix, *i.e.* it is the input object of some node. Three of its ports (o, s, r) are similar to the ports of c. You will notice that its only principal port is input i: p cannot interact until it has been instantiated with a channel in a communication. It also has an unblock port u that sends a signal to certain blocking (synchronization) machinery subjugated to the prefix.

Blocker node b is part of the blocking machinery. It will shortcut its data ports D_1 and D_2 when it receives an unblocking signal on its principal port U.

[7] This is not an interaction rule, but a multi-step reduction that will be enacted by appropriate interaction rules we will define.

[8] This term comes from (Parrow, 1995).

Unblocker node u will also shortcut its data ports d_1 and d_2 when it receives an unblocking signal, and will further unblock all blockers that are attached to its u port.

We use auxiliary nodes of the following types. They are used mostly as intermediate states in the evolution of main nodes, and thus serve to sequentialize such evolution. We don't describe their roles at this point, and will rather introduce them only in the interaction rules that they are involved in (⋈ 5.16, ⋈ 5.17, ⋈ 5.19). [⋈ 5.3]

⋈ 5.3: MIN_π Auxiliary Node Types.

5.2.2 MIN_π Labels

Instead of denoting channel and placeholder nodes with their node types c and p, we *label* the nodes with the π-calculus names that they correspond to. For example, we draw a c node labeled a as an oval containing a. The port types will always allow us to determine the node type unambiguously: the principal port of p is p.i, while c does not have an i port. We denote such a node in text as c[a].

If a name x is bound in a π-calculus process,[9] we place the label in parentheses, *e.g.* p[(x)], or place no label at all. We call such a label *hidden*. For brevity, we don't always parenthesize the labels of placeholder nodes; such labels should always be considered hidden.

The labels have several uses:

- They serve as a convenience for the reader.
- The incremental building of a net during the translation process uses the labels to decide which nodes to merge.
- They make finer distinctions between nets, so that *e.g.* the nets corresponding to $a!b$ and $a!c$ are considered different, even though they are graphically the same.

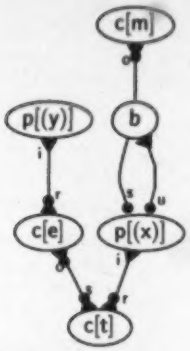
The labels play *no role* in the MIN_π reduction process. MIN_π reduction, like any IN reduction, is determined locally by principal ports, node types and interaction rules.

Definition 5.2.2.1 (Labeled MIN_π isomorphism) *Two MIN_π nets M and N are called isomorphic, $M \approx N$, if there is a graph isomorphism between them that also respects the node labels. Hidden/placeholder labels are excluded from this distinction.*

5.2.3 Example: The Telephone-Email Communication

Before we define our translation formally, we give the intended translation of the simple phone-email example 5.1: [⋈ 5.4]

[9] This includes the input object in $c?x.P$ and the hidden name in $(x)P$.



⋈ 5.4: Translation of the Telephone-Email Process $t?x.x!m, t!e, e?y$.

Here we have used the "heavy-weight" notation c[t] to denote a channel node marked with label t, and similarly for e, m, x, y. In the future we will denote nodes simply with the label (*e.g.* t and e, m, x, y), in order to avoid clutter.

The net above simply reflects the connectivity structure that is implicit in the corresponding π-calculus term. Every name is represented as a node. There is also a blocker node b that serves to prevent $x!m$ from acting until $t?x$ acts (this corresponds to the synchronization that is implied by the input prefix $t?x.x!m$).

You can understand the translation better if you read the π-calculus process and the net in parallel:

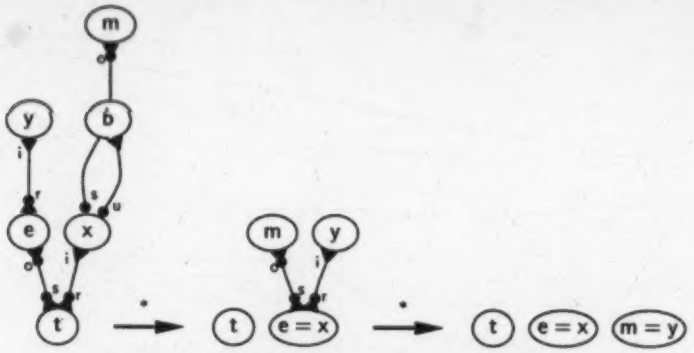
Π Part	Explanation	Network Fragment
$t?x$	A name x is received on channel t.	x is connected to t.r.
$x!m$	The name m is sent on channel x.	m is connected to x.s (indirectly thourgh a blocker b).
$t?x.x!m$	$x!m$ is prefixed by $t?x$, *i.e.* $x!m$ cannot act until $t?x$ acts and is reduced away. Thus $x!m$ is blocked.	A blocker b is inserted between x and m. The blocker b is controlled by the object of the prefix, x. Thus, the principal port of b is connected to the unblock port x.u.
$t!e$	The name e is sent on channel t.	e is connected to t.s.
$e?y$	A name y is received on channel e.	y is connected to e.r.

Table 5.2: Translation of the Telephone-Email Process.

The process reduces in this way:

$$t?x.x!m, t!e, e?y \rightarrow (e!m, e?y)[e/x] \rightarrow 0[e/x, m/y].$$

Here we have denoted with $P[x/y]$ the result of performing the substitution $[x/y]$ in the process P. In this particular case, the substitutions will have no effect, but we wanted to emphasize them in order to make a parallel with the corresponding MIN reduction: [⋈ 5.5]

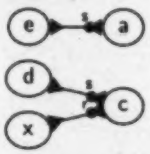


⋈ 5.5: Reduction of the Telephone-Email Process.

Here we have denoted with e = x and m = y the result of "merging" the nodes e, x and m, y respectively, embodied in the migration of the edges of x and y to e and m respectively. Each of the two multi-step reductions consists of three parts (§5.4; we illustrate the first multi-step reduction): *send/receive* in which t.s and t.r "choose" two nodes that are ready to interact and connect them directly, *input/output* in which the edges of p[x] are migrated to c[e] and *unblocking* in which the blockers pending on p[x] are removed.

5.2.4 Example: Blocking and Unblocking

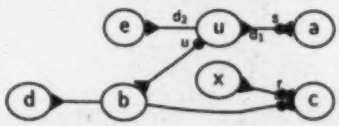
To clarify the role of the blocking machinery, we give another example. Consider the process $a!e.c!d, c?x$. The connectivity structure of this process, if we omit the blocking machinery (and in effect, treat the output prefix $a!e$ as an output *atom*), is as follows: [⋈ 5.6]



⋈ 5.6: Simplified Connectivity Structure of $[a!e.c!d, c?x]$.

In the above net, the subnet c−d corresponding to the output atom $c!d$ is ready to interact with the input atom $c?x$. However, it would be wrong to allow $c!d$ to interact in this way, because it is prefixed by $a!e$, and must therefore wait until $a!e$ is reduced away.

We remedy this by "breaking" the edge c−d and adding a blocker b. We also and an unblocker u controlled by a, so as to subjugate c−d to a−e: [⋈ 5.7]



⋈ 5.7: Adding a Blocker and Unblocker.

5.3 The Translation from π-calculus to MIN_π

Notation 5.3.0.1 *The sets of nodes and edges of a net N will be denoted as $n(N)$ and $e(N)$ respectively. We will also write $N = (n, e)$ for $n = n(N)$, $e = e(N)$.*

$n(M)+n(N)$ will denote the amalgamated sum *of two node sets $n(M)$ and $n(N)$, which is simply $n(M) \cup n(N)$ where nodes with the same label are identified.*

As an example of amalgamated sum, let $n(M) = \{c[q], c[r]\}$ and $n(N) = \{c[r], c[s]\}$. Then $n(M) + n(N) = \{c[q], c[r], c[s]\}$. Don't forget that on diagrams, when the node types are unambiguous, we would often write simply $n(M) = \{q, r\}$ and $n(N) = \{r, s\}$ for brevity.

We define a translation function $[\![\cdot]\!]: \Pi \mapsto \text{MIN}_\pi$ from π-calculus processes P to MIN_π nets N (*i.e.* $N = [\![P]\!]$). Simultaneously with it, we define an auxiliary set of *blocking points* $\mathcal{B}[\![P]\!]$, which are ("the middles of") some of the edges of $[\![P]\!]$: $\mathcal{B}[\![P]\!] \subset e([\![P]\!])$. The set $\mathcal{B}[\![P]\!]$ of a process P is defined from the sets $\mathcal{B}[\![P']\!]$ of its components P'. The set $\mathcal{B}[\![P]\!]$ is used by some steps of the translation (namely the two prefixes).

As mentioned in §5.2.2, c nodes of $[\![P]\!]$ bear labels throughout the translation process, but these labels do not influence the MIN_π reduction process.

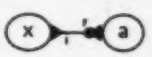
We define the translation by induction on the structure of P (§5.1), but make a distinction between atoms $\pi.0$ and proper prefixes $\pi.P'$.

Zero $[\![0]\!] = (\emptyset, \emptyset)$ and $\mathcal{B}[\![0]\!] = \emptyset$.

Parallel Composition $[\![P, Q]\!] = (n[\![P]\!] + n[\![Q]\!], e[\![P]\!] \cup e[\![Q]\!])$ and $\mathcal{B}[\![P, Q]\!] = \mathcal{B}[\![P]\!] \cup \mathcal{B}[\![Q]\!]$. Nodes of the same label are identified, and their edge sets are merged, but no edges are identified.

Hiding/restriction $[\![(c)P]\!] = [\![P]\!]\backslash c$, where $[\![P]\!]\backslash c$ is the same as $[\![P]\!]$, but the label of node c (if any) is erased, and therefore the node cannot be further identified with any other node in an amalgamated sum: the hidden label is treated as distinct from any other label. $\mathcal{B}[\![(c)P]\!] = \mathcal{B}[\![P]\!]$.

Input Atom $[\![a?x]\!]$[10]$= (\{\text{c[a]}, \text{p[(x)]}\}, \{\text{a.}r\text{–x.i}\})$ (channel, placeholder, and an edge connecting their r and i ports respectively). $\mathcal{B}[\![a?x]\!] = \{\text{a.}r\text{–x.i}\}$.[11] [⋈ 5.8]

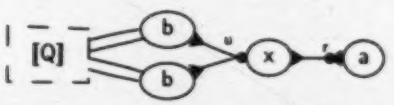


⋈ 5.8: Translation of π-calculus Input Atom: $[\![a?x]\!]$.

Output Atom $[\![a!x]\!]$[12]$= (\{\text{c[a]}, \text{c[x]}\}, \{\text{a.s–x.o}\})$ (two channels and an edge connecting their s and o ports). $\mathcal{B}[\![a!x]\!] = \{\text{a.s–x.o}\}$. [⋈ 5.9]

⋈ 5.9: Translation of π-calculus Output Atom: $[\![a!x]\!]$.

Input Prefix $[\![a?x.Q]\!]$ is defined from $[\![Q]\!]$ using the following additions/modifications:[13] [⋈ 5.10]



⋈ 5.10: Translation of π-calculus Input Prefix: $[\![a?x.Q]\!]$.

1. If a channel node labeled a is not present in $[\![Q]\!]$, then a new channel node a is added.

[10] More precisely, $[\![a?x.P]\!]$ where $P \equiv 0$.

[11] Remember that in order to avoid clutter, we often omit multiports with no edges from a figure.

[12] More precisely, $[\![a!x.P]\!]$ where $P \equiv 0$.

[13] Please imagine that a and x could be immersed in $[\![Q]\!]$. The figure would be precisely correct if that is not the case, and if in addition the size of $\mathcal{B}[\![Q]\!]$ is $|\mathcal{B}[\![Q]\!]|=2$ (notice that there are precisely two blockers b).

2. If a channel node labeled x (c[x]) is not present in $[\![Q]\!]$, then a new placeholder node labeled x (p[x]) is added.

   Otherwise, x's type is *demoted* from channel to placeholder:

   (a) The channel node c[x] is replaced with a placeholder node p[x].

   (b) Edges are reconnected as follows: c.s to p.s, c.r to p.r, c.o to p.o; c.c is empty since a main node c doesn't have any edges on c.c; p.u and p.i are left without any edges

   In this case, the placeholder p is effectively merged to a channel in $[\![Q]\!]$, in contrast to the Parallel Composition case above, which never amalgamates placeholders, since their labels are always hidden.

   For example, consider $[\![x!y]\!]$ which is a net with two nodes and one edge: c[x].s−c[y].o. If we prefix it with $a?x$, then $[\![a?x.x!y]\!]$ is a net where the type of node c[x] is changed to p[x] and which includes the following edges:[14] {c[a].r−p[x].i, p[x].s−c[y].o}.

3. A new edge a.r−x.i is added.

4. New blocker nodes b are inserted in every edge of $\mathcal{B}[\![Q]\!]$: for every $e = (n_1, n_2) \in \mathcal{B}[\![Q]\!]$, we add a blocker node b_e to the result; we remove e from the result and replace it with the two edges $\{n_1 - b_e.d_1, b_e.d_2 \stackrel{e}{-} n_2\}$. The principal ports of these b nodes are connected to p[x].u.

To summarize more formally, the nodes of $[\![a?x.Q]\!]$ are

$$n[\![a?x.Q]\!] = (n[\![Q]\!] + \mathsf{c[a]} + \mathsf{[x]}) \setminus\setminus \mathsf{x} \cup \bigcup_{e \in \mathcal{B}[\![Q]\!]} \mathsf{b}_e$$

where $n \setminus\setminus \mathsf{x}$ is like n but with the type of node [x] demoted from channel to placeholder and its label erased, and $\bigcup_{e \in \mathcal{B}[\![Q]\!]} \mathsf{b}_e$ denotes a set of blockers b indexed by edges e in the set $\mathcal{B}[\![Q]\!]$.

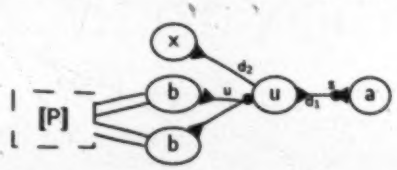
The edges of $[\![a?x.Q]\!]$ are

$$e[\![a?x.Q]\!] = e[\![Q]\!] - \mathcal{B}[\![Q]\!] \cup \{\mathsf{a}.r\text{−}\mathsf{x}.i\} \cup \bigcup_{e=(n_1,n_2) \in \mathcal{B}[\![Q]\!]} \{\mathsf{x}.u\text{−}\mathsf{b}_e, \mathsf{b}_e.d_1\text{−}n_1, \mathsf{b}_e.d_2\text{−}n_2\}$$

where '−' denotes set difference, and $\bigcup_{e=(n_1,n_2) \in \mathcal{B}[\![Q]\!]}$ again denotes union indexed by edges e in the set $\mathcal{B}[\![Q]\!]$, but we further let n_1 and n_2 denote the nodes at the two ends of e.

Finally, $\mathcal{B}[\![a?x.P]\!] = \{\mathsf{a}.r\text{−}\mathsf{x}.i\}$. The fact that this is a singleton set is a key to the linear nature of our translation (see 5.3.1).

Output Prefix $[\![a!x.P]\!]$ is defined similarly to input prefix, but an extra node u is added, because x may be the output subject of more than one prefix [15] [⋈ 5.11]



⋈ 5.11: Translation of π-calculus Output Prefix: $[\![a!x.P]\!]$.

1. New channel nodes a and x are added if not present in $[\![P]\!]$.

2. An unblocker node u is added.

[14] It also includes some blocking edges, see 4 below

[15] Please imagine that a and x could be immersed in $[\![P]\!]$. The figure would be precisely correct if that is not the case, and if in addition the size of $\mathcal{B}[\![P]\!]$ is $|\mathcal{B}[\![P]\!]|$=2 (notice that there are precisely two blockers b).

3. New edges $\mathsf{a.s}-\mathsf{u.d}_1$ and $\mathsf{u.d}_2-\mathsf{x.o}$ are added.
4. New blocker nodes b are inserted in every edge of $\mathcal{B}[\![P]\!]$ and their principal ports are connected to u.

More formally, the nodes of $[\![a!x.P]\!]$ are

$$n[\![a!x.P]\!] = (n[\![P]\!] + \mathsf{c[a]} + \mathsf{c[x]} + \mathsf{u}) \cup \bigcup_{e \in \mathcal{B}[\![P]\!]} \mathsf{b}_e.$$

The edges of $[\![a!x.P]\!]$ are

$$e[\![a!x.P]\!] = e[\![P]\!] - \mathcal{B}[\![P]\!] \cup \{\mathsf{a.s}-\mathsf{u.d}_1, \mathsf{x.o}-\mathsf{u.d}_2\} \cup \bigcup_{e=(n_1,n_2) \in \mathcal{B}[\![P]\!]} \{\mathsf{u.u}-\mathsf{b}_e, \mathsf{b}_e.\mathsf{d}_1-n_1, \mathsf{b}_e.\mathsf{d}_2-n_2\}.$$

And finally, the blocking set is $\mathcal{B}[\![a!x.P]\!] = \{\mathsf{a.s}-\mathsf{u.d}_1\}$. The fact that this is a singleton set is a key to the linear nature of our translation (see §5.3.1).

Example We now go back to ⋈ 5.4 and give the translation of that example step-by-step, corresponding to the formation rules given above.

1. The output atom $x!m$ is translated to two nodes and an edge: $\mathsf{c}[x].\mathsf{s}-\mathsf{c}[m].\mathsf{o}$.
2. While translating the input prefix $t?x$, the type of x is demoted from channel to placeholder, so it becomes $\mathsf{p}[x]$.
3. The edge $\mathsf{p}[x].\mathsf{s}-\mathsf{c}[m].\mathsf{o}$ is "broken" since it's in $\mathcal{B}[\![x!m]\!]$, and a blocker b is inserted in it.
4. The principal port of b is connected to $\mathsf{p}[x].\mathsf{u}$.
5. The input prefix $t?x$ itself is translated simply to a channel $\mathsf{c}[t]$ and an edge $\mathsf{c}[t].r-\mathsf{p}[x].\mathsf{i}$.
6. The input atom $e?y$ is translated to two nodes and an edge: $\mathsf{c}[e].r-\mathsf{p}[y].\mathsf{i}$.
7. The output atom $t!e$ is translated to two nodes and an edge: $\mathsf{c}[t].\mathsf{s}-\mathsf{c}[e].\mathsf{o}$.
8. The three parallel components of $[\![t?x.x!m, t!e, e?y]\!]$ are amalgamated. In this process the different copies of t and e that were introduced in previous steps are merged to one copy each, and that is how t and e get to have two edges each.

5.3.1 Complexity of the Translation

We now prove that our translation has linear complexity, both in terms of space (number of nodes) and time (number of reductions).

Proposition 5.3.1.1 (Structural Complexity) *The size $|[\![P]\!]|$ of $[\![P]\!]$ (number of nodes plus number of edges) is linear (or sub-linear) in the size $|P|$ of P (number of prefixes).*

Proof We will prove by structural induction the auxiliary lemma that $|[\![P]\!]| + |\mathcal{B}[\![P]\!]| \leq 4\,|P|$:

Step	(In)equality	Justification
Zero	$\|[\![0]\!]\| + \|\mathcal{B}[\![0]\!]\| = 0 = 4 \times 0 = 4\,\|0\|$	by construction and the definition of Zero
Parallel	$\|[\![P,Q]\!]\| + \|\mathcal{B}[\![P,Q]\!]\|$ $\leq (\|[\![P]\!]\| + \|[\![Q]\!]\|) + (\|\mathcal{B}[\![P]\!]\| + \|\mathcal{B}[\![Q]\!]\|)$	by construction; an inequality since amalgamated sum may merge some nodes

Step	(In)equality	Justification
	$\leq 4\|P\| + 4\|Q\|$	by induction hypothesis
	$= 4\|(P,Q)\|$	by definition of Parallel in π-calculus
Hiding	$\|[(c)P]\| + \|\mathcal{B}[(c)P]\|$	
	$= \|[P]\| + \|\mathcal{B}[P]\|$	by construction
	$\leq 4\|P\| = 4\|(c)P\|$	by induction hypothesis and definition of Hiding
Input Atom	$\|[a?x]\| + \|\mathcal{B}[a?x]\| = 2 + 1$	by construction
	$\leq 4 = 4 \times 1 = 4\|a?x\|$	by definition of prefix/atom
Output Atom	$\|[a!x]\| + \|\mathcal{B}[a!x]\| = 2 + 1$	by construction
	$\leq 4 = 4 \times 1 = 4\|a!x\|$	by definition of prefix/atom
Input Prefix	$\|[a?p.Q]\| + \|\mathcal{B}[a?p.Q]\|$	
	$\leq (2 + \|[Q]\| + \|\mathcal{B}[Q]\|) + \|\mathcal{B}[a?p.Q]\|$	by construction; it is an inequality since a and/or p may be already present in $[Q]$
	$\leq 2 + 4\|Q\| + 1$	by induction hypothesis and the key fact $\|\mathcal{B}[a?p.Q]\| = 1$
	$< 4(\|Q\| + 1) = 4\|a?p.Q\|$	by definition of prefix
Output Prefix	$\|[a!c.P]\| + \|\mathcal{B}[a!c.P]\|$	
	$\leq (3 + \|[P]\| + \|\mathcal{B}[P]\|) + \|\mathcal{B}[a!c.P]\|$	by construction; it is an inequality since a and/or c may be already present in $[P]$
	$\leq 3 + 4\|P\| + 1$	by induction hypothesis and the key fact $\|\mathcal{B}[a!c.P]\| = 1$
	$= 4(\|P\| + 1) = 4\|a!c.P\|$	by definition of prefix

Table 5.3: Proof of the Linear Complexity of Translation $\Pi \rightarrow$ MIN.

◊

We also note in passing that $|\mathcal{B}[P]|$ is equal to the number of top-level parallel components of P: if $P = (x_1 \ldots x_m)P_1 \ldots P_n$ where $x_1 \ldots x_m$ are all name restrictions pushed to top-level, and every $P_i = \pi_i.P_i'$ is a prefixed process, then $|\mathcal{B}[P]| = n$. The reason is simply that $|\mathcal{B}[P_i]| = 1$ for every prefixed or atomic process P_i.

Corollary 5.3.1.2 (Time Complexity) *The number of steps a MIN reduction takes to emulate a corresponding* Π *reduction is linear in the length of the* Π *reduction.*

Proof An obvious corollary of 5.3.1.1 and the fact that our MIN reductions always decrease the number of nodes (see §5.5 for insights into this). ◊

5.3.2 Structural Correspondence of the Translation

Notation

We now check statically that our translation neither identifies processes that are not structurally equivalent, nor fails to identify equivalent processes.

Theorem 5.3.2.1 $[P] \approx [Q]$ *iff* $P \equiv Q$ *(*$\approx$ *as per Definition 5.2.2.1).*

Proof **If** Case analysis on the definition of structural equivalence in §5.1.2.

1. α-renaming. Since the bound labels of P are removed in $[P]$, we have $[P] \approx [P[y/x]]$ for any name x not free in P.

2. Monoidal structure of parallel composition. Easy, due to $[0] = (\emptyset, \emptyset)$ and the associativity and commutativity of amalgamated sum $n_1 + n_2$ and union $e_1 \cup e_2$.

3. – 5. are obvious.

Only if We build an inverse translation $\langle\!\langle N \rangle\!\rangle$ and we prove that it preserves structural congruence (see below). ◊

5.3.3 Inverse Translation

We call a net N *main* or *primary* if it is the translation of a process, $N = [P]$. We call nets that don't correspond to a process *auxiliary* or *secondary*. (In this chapter we will only consider auxiliary nets that are the reducts of primary nets.)

We define an inverse translation $\langle\!\langle \cdot \rangle\!\rangle : \mathrm{MIN}_\pi \mapsto \Pi/_\equiv$ from a primary net N to a representative of the structural equivalence class of the original process, $P = \langle\!\langle N \rangle\!\rangle$. The translation breaks a composite net into parallel components, strips prefixes and associated blocking machinery, and works recursively.

As a preliminary step, we take care of hidden nodes: $\langle\!\langle N \rangle\!\rangle = (c_1 \ldots c_n)\langle\!\langle N' \rangle\!\rangle$ where $c_1 \ldots c_n$ are all the c nodes of N with hidden labels, and N' is like N but with these labels un-hidden.

We are careful to define $\langle\!\langle N \rangle\!\rangle$ to be a function, not a relation (to be "deterministic"), Therefore, we first find exhaustively all parallel components of N.

Definition 5.3.3.1 (Blocking Regions) *A* blocking region b *of a main net* N *is a minimal non-empty set of directed edges* $e \subset e(N)$ *closed under a "principal domination" condition: if* $\mathsf{y.q{-}x.p} \in e$ *is an edge of* N *that comes into the single principal port* p *of node* x[16] *then* $\mathsf{x.p_i{-}y_i.q_i} \in e$ *for all other edges of* x.

For example on ⋈ 5.4, one blocking region is {c[t].r−p[x], p[x].s−b, p[x].u−b, b−c[m].o} (from c[t] we enter p[x] through its single principal port, and from there we enter b through its single principal port). This net has two more blocking regions, both of which are singletons: {c[t].s−c[e].o} and {c[e].r−p[y].i}.

By minimality, a blocking region b either consists of one edge, both ends of which are not single-principal ports, or is a tree of edges related by principal domination. In other words, b cannot contain unrelated edges. The possible edges of b are

input prefix	$\mathsf{c.r{-}p.i}, \mathsf{c.r{-}b.d_1}, \mathsf{b.d_2{-}p.i}$
output atom	$\mathsf{c.s{-}c.o}, \mathsf{c.s{-}b.d_1}, \mathsf{b.d_2{-}c.o}$
output prefix	$\mathsf{c.s{-}u.d_1}, \mathsf{u.d_2{-}c.o}, \mathsf{c.s{-}b.d_1}, \mathsf{b.d_2{-}u.d_1}$
blocking control	$\mathsf{u.u{-}b.u}, \mathsf{p.u{-}b.u}$

Table 5.4: Possible Edges in a Blocking Region

Lemma 5.3.3.2 (Disjointness of blocking regions) *Two blocking regions* b_1 *and* b_2 *are either disjoint or one is a subset of the other.*

[16] This means that x is one of p, b, u but not c, since c has several principal ports.

Proof Assume the opposite. Since b_1 and b_2 are trees, by following non-shared edges we can reach a shared edge. The source node x of that shared edge must have two incoming non-shared edges, one in b_1 and the other in b_2. But this is impossible, because x cannot have multiple principal ports, nor principal multiports. ◊

Lemma 5.3.3.3 (A blocking region is a main net) *Let M be the subgraph of N generated by a blocking region b of N (M contains all edges in b and their adjacent nodes). Then M is a main net.*

Proof Follows from the principal domination closeness of b, after observing that all edges of $[\![P]\!]$ in $[\![a!c.P]\!]$ and $[\![a?p.P]\!]$ can be reached from a.s−u.d_1 and a.r−p.i respectively (see ⋈ 5.10 and ⋈ 5.11). This observation can be proved easily by induction on the structure of $[\![P]\!]$. ◊

Let $B = \{b_1, \ldots, b_n\}$ be the set of all maximal blocking regions of N. By Lemma 5.3.3.2 and maximality, they partition the edges of N: $e(N) = b_1 + \cdots + b_n$. By Lemma 5.3.3.3, they all are main nets: $b_i = [\![N_i]\!]$. So we define the first step of our inverse translation (breaking into parallel components) thus: if N has more than one maximal blocking regions $\{b_1, \ldots, b_n\}$, then

$$\langle\!\langle N \rangle\!\rangle \stackrel{\text{def}}{=} \langle\!\langle N_1 \rangle\!\rangle, \ldots, \langle\!\langle N_n \rangle\!\rangle$$

where N_i is the subgraph of N generated by b_i.

To define the inverse translation of a maximal ("main") blocking region b, we have to consider two cases:

1. b consists of a single edge. The edge can be c[a].s−c.o or c[a].r−p.i. Then $\langle\!\langle b \rangle\!\rangle \stackrel{\text{def}}{=} a!c$ or $\langle\!\langle b \rangle\!\rangle \stackrel{\text{def}}{=} a?p$ respectively.

2. b is a tree of several edges. Then it has a single root edge, for otherwise the root edges would be unrelated. Furthermore, b must correspond to ⋈ 5.10 or ⋈ 5.11, because it is main (subgraph of a main net $N = [\![R]\!]$).

   (a) The root edge is c[a].s−u.d_1. Then $\langle\!\langle b \rangle\!\rangle \stackrel{\text{def}}{=} a!c.\langle\!\langle P_1 \rangle\!\rangle$ where P_1 is the subnet connected to the b nodes (labeled $[\![P]\!]$ in the figure).

   (b) The root edge is c[a].r−p.i. Then $\langle\!\langle b \rangle\!\rangle \stackrel{\text{def}}{=} a?p.\langle\!\langle Q_1', Q_2' \rangle\!\rangle$ where Q_1 is the subnet connected to the b nodes (labeled $[\![Q]\!]$ in the figure), Q_2 are optional subtrees of b attached to the s and r ports of p, and Q' is like $\langle\!\langle Q \rangle\!\rangle$, but with the type of p promoted to c.

The proof of Theorem 5.3.2.1 is completed by the following lemma.

Lemma 5.3.3.4 *If two main nets are isomorphic, $M \approx N$, then $\langle\!\langle M \rangle\!\rangle \equiv \langle\!\langle N \rangle\!\rangle$.*

Proof By isomorphism, the blocking regions of M and N are also isomorphic. Then by the determinism of $\langle\!\langle \cdot \rangle\!\rangle$, it follows that $\langle\!\langle M \rangle\!\rangle$ and $\langle\!\langle N \rangle\!\rangle$ are essentially the same, up to α-renaming of hidden labels. ◊

5.3.4 Example: The Honda-Tokoro Construction

In order to illustrate our translation, we present a substantial example, the Honda-Tokoro construction. This is a classical construction in the theory of the π-calculus, implementing polyadic (n-ary) blocking prefix in terms of monadic (1-ary) output atoms and monadic input blocking prefix. It is also known as the *zipper construction*, because of the manner in which output atoms and input prefixes interlock to constrain and control the reduction. For $n = 2$, the construction is defined as follows:

$$\begin{aligned} z![x_1 x_2].P &= (m)z!m, m?p_1.(p_1!x_1, m?p_2.(p_2!x_2, P)) \\ z?[y_1 y_2].Q &= (c_1 c_2)z?n.(n!c_1, c_1?y_1.(n!c_2, c_2?y_2.Q)) \end{aligned}$$

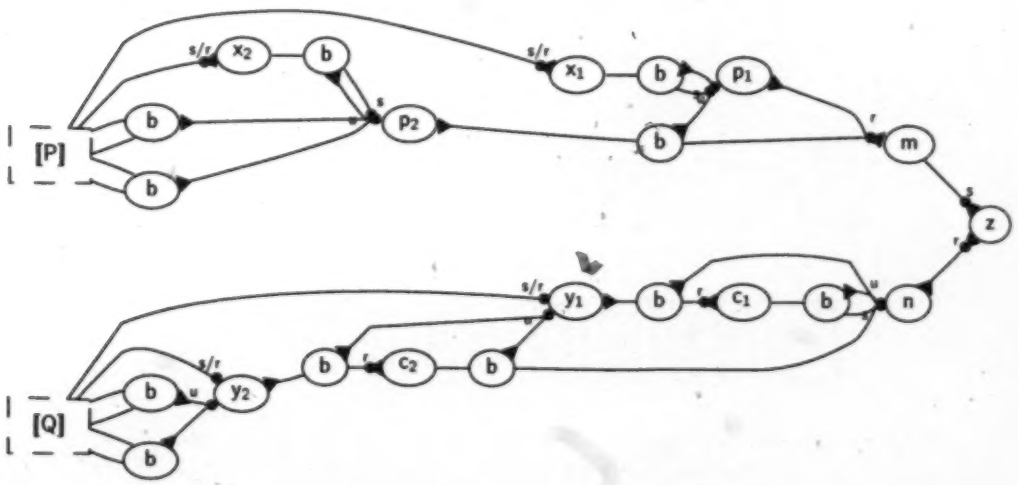
where m, n, c_1, p_1, c_2, p_2 are fresh names that don't appear in P, Q.

Here is how the Honda-Tokoro construction serves its role of n-ary prefix:

	$z![x_1x_2].P, z?[y_1y_2].Q$
$\equiv$	$(\nu c_1c_2)z!m, m?p_1.(p_1!x_1, m?p_2.(p_2!x_2, P)), z?n.(n!c_1, c_1?y_1.(n!c_2, c_2?y_2.Q))$
$\xrightarrow{z!m,z?n}$	$(\nu c_1c_2)m?p_1.(p_1!x_1, m?p_2.(p_2!x_2, P)), (m!c_1, c_1?y_1.(m!c_2, c_2?y_2.Q))$
$\xrightarrow{m?p_1,m!c_1}$	$(\nu c_1c_2)(c_1!x_1, m?p_2.(p_2!x_2, P)), c_1?y_1.(m!c_2, c_2?y_2.Q))$
$\xrightarrow{c_1!x_1,c_1?y_1}$	$(\nu c_1c_2)m?p_2.(p_2!x_2, P)), (m!c_2, c_2?y_2.[x_1/y_1]Q))$
$\xrightarrow{m?p_2,m!c_2}$	$(\nu c_1c_2)(c_2!x_2, P)), c_2?y_2.Q[x_1/y_1]))$
$\xrightarrow{c_2!x_2,c_2?y_2}$	$(\nu c_1c_2)P, Q[x_1/y_1, x_2/y_2]$
$\equiv$	$P, Q[x_1/y_1, x_2/y_2]$

Table 5.5: Reduction of the Honda-Tokoro Construction.

The translation $[\![z![x_1x_2].P, z?[y_1y_2].Q]\!]$ is given below. Note that we don't need any u nodes since there are no output prefixes. On this figure, please imagine that x_1, x_2 are embedded in $[\![P]\!]$ and y_1, y_2 are embedded in $[\![Q]\!]$ (we have no easy way of depicting them that way with the graph layout software that we use). We have marked one of the ports of x_1, x_2, y_1, y_2 with s/r, to symbolize that these nodes can have their s or r or both ports connected to other nodes of $[\![P]\!]$ and $[\![Q]\!]$, depending on whether these nodes serve as input, output or "both" subjects in $[\![P]\!]$ and $[\![Q]\!]$. [⋈ 5.12]



⋈ 5.12: Translation of the Honda-Tokoro Construction.

Now we trace a few steps of the translation of this construction. We start from the innermost part of $[\![z![x_1x_2].P]\!]$:

1. $[\![p_2!x_2]\!]$ consists of the two nodes p_2 and x_2, which are directly connected by an edge. $\mathcal{B}[\![p_2!x_2]\!]$ consists of that edge.
2. The prefix $[\![m?p_2]\!]$ inserts blockers in every edge of $\mathcal{B}[\![p_2!x_2, P]\!]$. This includes the edge p_2–x_2, as well as (by assumption) two edges of $[\![P]\!]$; all three blockers are connected to the unblocking port $p_2.u$.
3. The prefix $m?p_2$ itself is translated to the nodes m and p_2 and a direct edge between them. The blocking set of the translation thus far consists of that edge.

4. $[p_1!x_1]$ consists of the two nodes p_1 and x_1, which are directly connected by an edge. $\mathcal{B}[p_1!x_1]$ consists of that edge p_1–x_1. When we add it to the blocking set m–p_2 thus far, we have two blocking points (corresponding to the two rightmost blockers in the upper half of the figure).

5. The prefix $[m?p_1]$ inserts these two blockers and connects them to $p_1.u$. The prefix itself is translated to the node m (node p_1 is already in the translation) and the edge between them. The blocking set of the translation thus far consists of that edge, but no blocker is ever inserted in it, since it is not subject to a prefix.

6. Finally, $z!m$ adds another node z and an edge z–m.

The translation of the other half $[z?[y_1y_2].Q]$ is similar.

Apart from the blocking machinery needed to sequentialize the interactions $p_i \bowtie c_i$ $(i = 1, 2)$ on the shared channel $m \bowtie n$, the construction is symmetric.[17]

5.3.5 Unnecessary Blocking in the π-calculus

Due to the linear nature of π-calculus terms, one is forced to introduce more blocks than are really necessary. Let's consider the *dependency relation* $\prec$ between prefixes in a process P, the intention being that $\pi_1 \prec \pi_2$ means that π_1 must be reduced away before π_2 can be reduced. We define $\prec$ as the transitive closure of the following rules:

1. If $\pi.Q$ is a subterm of P then $\pi \prec \pi'$ for every π' occurring in Q. (Blocking; explicit synchronization.)

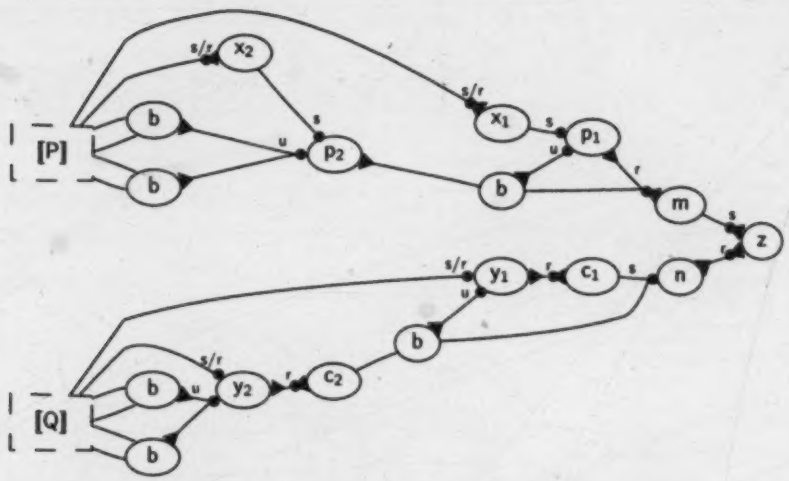
2. If a prefix $a?x$ si in P, then $a?x \prec \{b!x, x!b, x?b\}$[18] for all prefixes and atoms in P that mention x. Remember, we assumed that all bound names are renamed-apart. (Provision; data-dependency-based synchronization.)

Since the π-calculus only allows disjoint or properly nested scopes but not overlapping scopes, the structure of $\prec$ is a directed forest. For example, it is impossible to have $\pi_1 \prec \pi$ and $\pi_2 \prec \pi$ without also having $\pi_1 \prec \pi_2$ or $\pi_2 \prec \pi_1$. Furthermore, since the scopes of input prefix and synchronization are confounded, 1 subsumes 2.

There are no such limitations in our MIN_π representation, because it features localized and distributed blocking. This allows us to minimize the use of blocks. For example, the only blocks that are necessary in the Honda-Tokoro construction $\bowtie$ 5.12 are blocks that enforce $m?p_1 \prec m?p_2$, $n!c_1 \prec n!c_2$, $p_2!x_2 \prec P$ and $c_2?y_2 \prec Q$. The other blocks are subsumed by provision ("natural" dependency). Therefore, we can easily remove 5 blocks. [$\bowtie$ 5.13]

[17] Please note that since these nets are the translation of π-calculus processes, they contain only main nodes and no auxiliary nodes. We ran out of letters, that is why we used some subscripts. Here z, m, c_1, c_2, x_1, x_2 are channels, and n, p_1, p_2, y_1, y_2 are placeholders.

[18] Here $\pi \prec S$ means $\pi \prec \pi'$ for every $\pi' \in S$.



⋈ 5.13: Minimized Variant of the Honda-Tokoro Construction.

5.4 MIN_π Interaction Rules

We now describe the interaction rules that govern MIN_π. Communication is implemented in three discrete steps, corresponding to the different roles of prefix that we identified in §5.1.4:

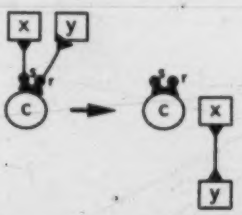
Send/Receive A pair of s/r links of a channel c is chosen non-deterministically, and the objects at the ends of these links (say x in $a!x$ and y in $a?y$) are put in contact.

Input/Output (Link Migration) All links of the placeholder p are transferred to the channel c that is instantiating it.

Unblocking The processes that were blocked by the two prefixes (*e.g.* P and Q in $a?p.P$ and $a!c.Q$) are unblocked and can interact further.

5.4.1 Send/Receive

In the first stage of communication, a channel c that has *active*[19] nodes attached to both its s and r ports, short-circuits them in order to make it possible for them to interact. MIN_π does not give us easy means to secure two active nodes atomically. The easiest would be to employ a ternary rule[20] [⋈ 5.14]

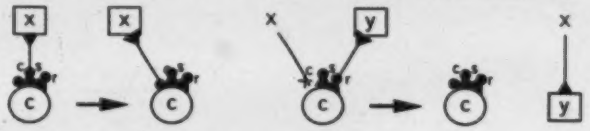


⋈ 5.14: (Hypothetical) Ternary Send/Receive Rule.

[19] Linked to c through their principal port.
[20] Remember that boxes denote "abstract" node types.

However, IN and MIN_π allow only binary rules.

Therefore we have to implement send/receive in two separate sub-steps, which may be distant in time. In the first sub-step channel c secures an active node from port s and stores it in the set of committed output objects on port c. In the second sub-step it puts an active node from port r in contact with an edge from port c. Both edges are selected non-deterministically. [⋈ 5.15]



⋈ 5.15: Send/Receive Rules.

Note An alternative to the above implementation would be to use an "intelligent" channel that knows the current arities (number of edges) on its s/r ports, like we did for our INMC→INMP translation (§4.7). Such a channel would not commit to ("buy into") a c.s ⋈ c interaction unless it is sure that there is a c.r ⋈ p interaction ready, so that it can complete the send/receive.

Notation We say that a node x is *active towards* a node y if x is connected to y by its principal port (x has its principal port "turned towards" y). Similarly, we say that a node x on port y.p is *active*, if the principal port of x is connected to y.p.

The correctness of this implementation depends on the following

Lemma 5.4.1.1 (Monotonicity of Activation) *Once a node on* c.s *becomes active, it cannot become inactive until it is transferred to* c.c*, and later detached from* c *by the above rule.*

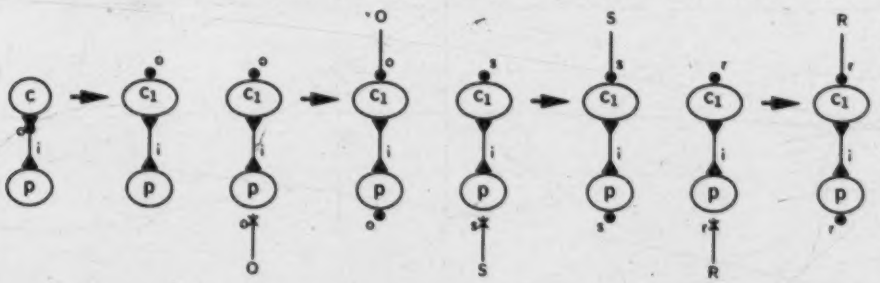
Proof Examination of the possible node types and applicable rules later in this section. The possible nodes on c.s in a translation $[\![P]\!]$ are $u.d_1$ (active), c.o (active) and b (inactive). The rules that affect c.s are immigration of links from p.s (see §5.4.2) which are again nodes of the same types, and the removal of b during unblocking (see §5.4.3) which exposes an active $u.d_1$ or c.o. Once a link is active, it can only be transferred to c.c where it will remain active until it is detached by the second rule above. ◇

5.4.2 Input/Output (Link Migration)

The second step of a communication $a!c.P, a?p.Q$ is the merging of the output object node c and the input object node p, corresponding to the π-calculus substitution $Q[c/p]$ (c is instantiated for p). In MIN_π, this entails a migration of all edges of p to c. Since c may be output object of more than one output prefix, if it keeps its type c during the immigration process, it may be subjected to another send/receive interaction while the first one is in progress.

Then c may immigrate links from another placeholder p', and therefore will interleave link immigration from two different placeholders. Such interleaved immigration increases the parallelism of the implementation, and should therefore be seen as a positive aspect. However, in this chapter we prefer to disallow it, in order to simplify the proof of correctness of our translation.

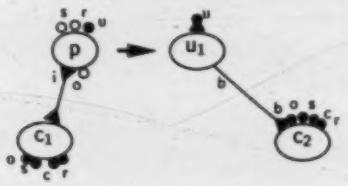
To this end, c changes its type to c_1 throughout the immigration process. c_1 is committed to that process and cannot perform any other interaction until the process is completed. [⋈ 5.16]



⋈ 5.16: Input/Output Rules.

Here we use arity constraints (see §3.3.2) to guide the migration process. For example, the second rule says that while there are edges on p.o, they should be moved one-by-one to c_1.o, and similarly for s and r edges.

After migration is completed, the placeholder p becomes an unblocker u_1 and c becomes c_2, waiting for a signal from u_1 that unblocking is completed (described in §5.4.3). [⋈ 5.17]



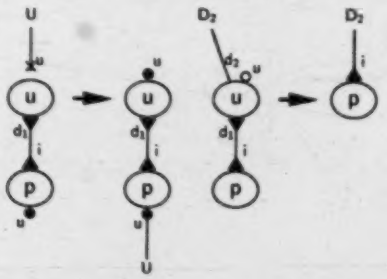
⋈ 5.17: Final Input/Output Rule.

Lemma 5.4.2.1 (Confluence of Link Migration) *1. The rules of this subsection* ⋈ 5.16 *and* ⋈ 5.17 *are confluent (as formulated in §2.3.1). 2. There is no outside interference, i.e. after the initial interaction* c ⋈ p *and until the final interaction* c_1 ⋈ p, *no other nodes but the ones mentioned in this subsection can participate.*

Proof Simple. Both c_1 and p have a single principal port, and any two link migration rules commute. ◊

5.4.3 Blocking and Unblocking

The send/receive rules of §5.4.1 are insensitive with respect to the type of the object, therefore they apply equally well to c ⋈ p (corresponding to an output atom $a!c$ interacting with an input $a?p.Q$), and to u ⋈ p (corresponding to an output prefix $a!c.P$ interacting with an input $a?p.Q$).[21] The case c ⋈ p was described above in §5.4.2. In the case u ⋈ p, u migrates all its blocking links to p (by the first rule below), and then disappears (by the second rule): [⋈ 5.18]

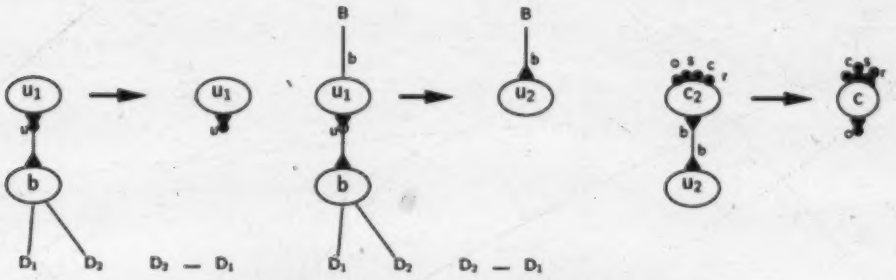


⋈ 5.18: Unblocking (step 1): Link Migration.

This effectively merges two edge bundles on port u, one on p.u which port holds blockers subjugated to the input prefix, and another on u.u which port holds blockers subjugated to the output prefix (see again the definition of the translations of input and output prefix in §5.3). Once this merging is completed, a c ⋈ p redex results, and the rules of §5.4.2 apply.

[21] Note that if we limit our consideration to the ν (asynchronous π) calculus where output prefix is not allowed, then we don't need u.

To complete the communication we must unblock all blockers b controlled by the prefix. More specifically, the node u_1 that was borne by the placeholder p in §5.4.2 should dismiss all the blockers that it controls. [⋈ 5.19]



⋈ 5.19: Unblocking (step 2): Removing Blockers.

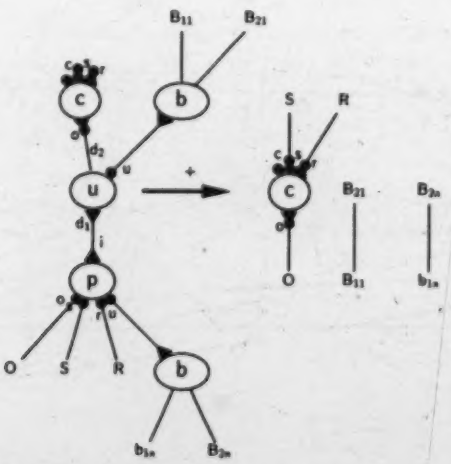
At the end of the unblocking process (directed by the arity constraint on the principal port), u_1 turns into u_2 and signals c_2 that the communication has been completed.

Lemma 5.4.3.1 (Confluence of Unblocking) *1. The rules of this subsection* ⋈ 5.18 *and unblock are confluent. 2. There is no outside interference.*

Proof Simple examination. Only u_1 has a principal multiport, but the order in which it dismisses b nodes is irrelevant. ◊

The lemmas in this and the previous subsection mean that after the non-deterministic selection of a c.o link, computation can proceed in essentially only one way, giving us the following aggregate result.

Lemma 5.4.3.2 *In the figure below (with any number of* b *nodes attached to* u *and* p*, and any number of* O, S, R *links), the* MIN_π *on the LHS can reduce in only one way, to the* MIN_π *on the RHS.* [⋈ 5.20]

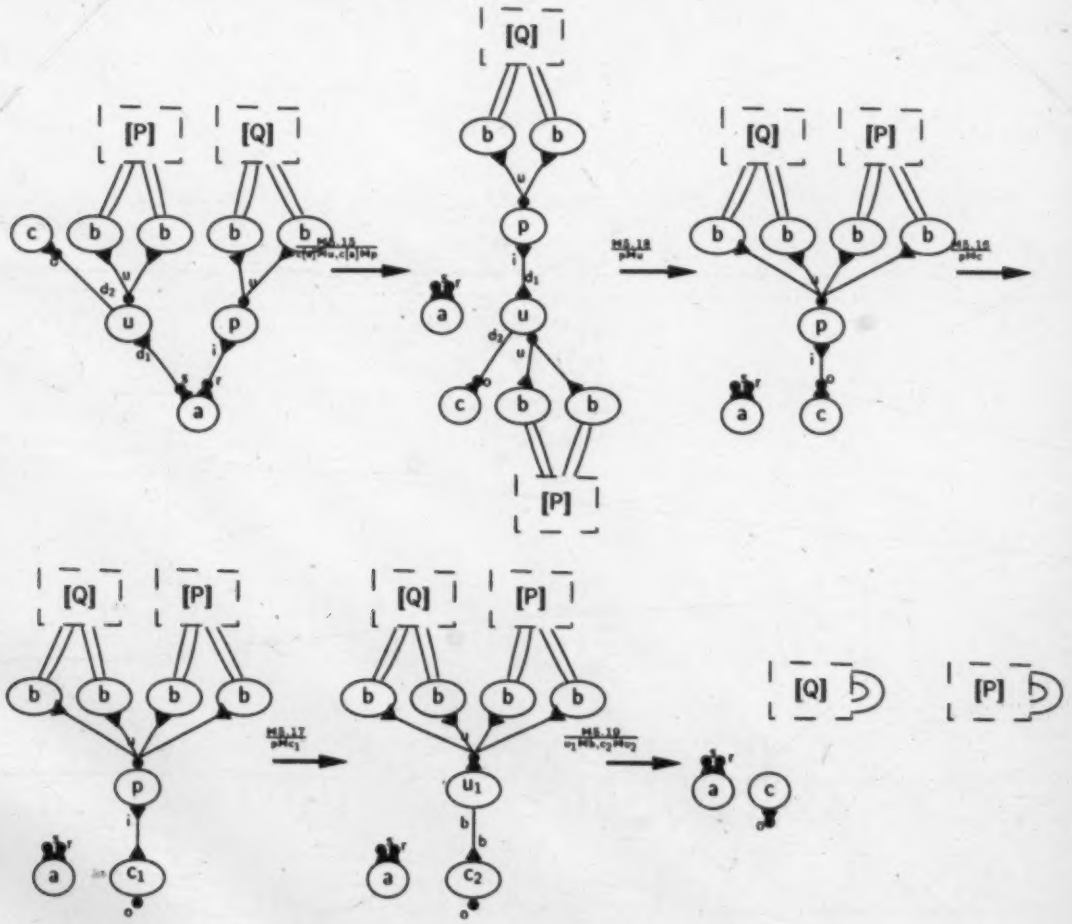


⋈ *5.20: Net (Aggregate) Result of* u ⋈ p *Interaction.*

This completes the description of the MIN_π interaction rules.

5.4.4 Example: Reduction of Prefix

As an example of the application of MIN_π rules, we give below a detailed trace of the reduction of $[\![a?p.Q, a!c.P]\!]$. As in ⋈ 5.10 and ⋈ 5.11, we assume that the names a, p, c don't occur in P, Q, *i.e.* that the nodes a, p, c don't occur in $[\![P]\!]$ and $[\![Q]\!]$. We also assume that $|\mathcal{B}[\![P]\!]| = |\mathcal{B}[\![Q]\!]| = 2$, that is why below there are two b nodes per prefix. [⋈ 5.21]



⋈ 5.21: The Reduction of $[\![a?p.Q, a!c.P]\!]$.

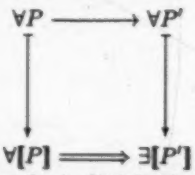
5.5 Operational Correspondence

We now set out to prove that our translation serves its purpose of faithfully modeling the π-calculus. Please first re-consult our general discussion about completeness and soundness of implementation translations in §4.1.1.

Completeness is easy, as is to be expected of any decent implementation encoding.

Theorem 5.5.0.1 (Completeness) *For every single-step process reduction $P \to P'$ there exists*

a corresponding multi-step net reduction $[P] \Rightarrow [P']$.



Proof Case analysis on the reduction rules in §5.1.3. For Comm, we use Lemma 5.4.3.2: starting from the net in §5.3, after the s and r links on a are committed, there is a (unique) reduction which removes all b and migrates the links of p to c, which is the same as $[P, Q[c/p]]$. For Struc, we need Theorem 5.3.2.1. Par and Res (top-level name restriction) do not decrease the possibilities for MIN_π reduction. ◊

5.5.1 Soundness

As is usual with implementation encodings, Soundness presents a much greater challenge. To make our task a bit easier, we have gone to great lengths to make the reduction process as deterministic as possible. For example in §5.4.2, we could increase the degree of parallelism by leaving c with the same type throughout the process, instead of sequentializing it using c_1 and c_2. However, until some general results about partial confluence of MINs are available (or coinductive techniques for IN are better understood, (Fernández and Mackie, 1998)), we prefer to simplify the reduction process in order to obtain more easily our correspondence result.

Uncommitting Precommitments

First, we need to resolve a simple technical difficulty. Our send-receive rule ⋈ 5.15 involves a pre-commitment step in the sense of §4.1.1 and §4.1.1, namely the transfer of an active edge from c.s to c.c. Take the simple net corresponding to c!v. In that net v is attached to c.s. The pre-commitment moves v to c.c, but this is not a primary net anymore (is not the translation of a process). And since there is no possibility for communication, this net cannot evolve at all, and will remain non-primary. We therefore introduce a simple mechanism to back-up pre-commitments, in order to be able to recognize such "quasi-primary" nets.

Definition 5.5.1.1 *Given a net M, its* uncommitment *N is obtained by transferring all links from* c.c *to* c.s, *for all channels* c. *We denote this function as $M \prec N$. We also define*

$$\overset{\prec}{\Rightarrow} \overset{\text{def}}{=} \Rightarrow \prec$$

.

Note that uncommitment is an operation external to MIN_π; a net itself can never back-up a commitment. There are no interaction rules permitting this, nor there is any need for it. Uncommitment is an auxiliary mechanism that helps us evaluate reducts of a primary net, in order to relate them back to processes.

Computing uncommitment is a decidable and tractable process. It takes linear time with respect to the size of a net. Therefore, we consider *Reduces*$[\prec]$ to be just "as good as" $\Rightarrow$ for the purpose of establishing operational correspondence.

Completing Partial Reductions

Take a primary net $[P]$, being the translation of a process P. Take a net N being a reduct of $[P]$: $[P] \Rightarrow N$. Before we can relate N back to a process, we need to complete net reductions forward to a "checkpoint". We cannot hope to arrange the checkpoint to be at the next single reduction step of P, say $P \rightarrow P'$. There are two reasons:

- N may have already reduced past that point.
- N may have interleaved parts from several independent single-step reductions of P.

The first reason can be eliminated by looking more comprehensively at net reducts, so that we don't miss imtermediate reducts.

The second reason however is inherent to our translation, and cannot be eliminated easily. Unless we impose some very restrictive reduction *strategies* on net reduction, we cannot force nets not to interleave single-step process reductions. But it would be unwise to limit the degre of parallelism of our implementation. This is a typical situation when implementing an "atomic" reduction relation in a completely distributed system.

Definition 5.5.1.2 *Given a net M, its* completion *N is obtained by firing all possible rules of §5.4.2 and §5.4.3. We denote this as $M \rhd N$.*

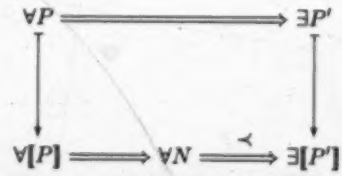
Lemma 5.4.3.2 guarantees that the completion of a net is unique.

Both $\rhd$ and $\prec$ are deterministic (they "don't make any decisions"). The former forges ahead by completing all communications that have been committed, the latter removes commitments to allow us to evaluate (read-back) the result. While $\prec$ is an operation external to the reduction system, $\rhd$ consists of reductions that the system is capable of performing itself, the completion operation simply forces those reductions.

The Soundness Theorem

We state the following

Theorem 5.5.1.3 (Soundness) *For every multi-step reduction $[P] \Rightarrow N$ there exists a "completing reduction" $N \stackrel{\prec}{\Rightarrow} [P']$ such that $P \Rightarrow P'$.*



The theorem states that every net reduction starting from a primary net $[P]$ can always be completed[22] to a primary net $[P']$ such that $P \Rightarrow P'$. In other words, reducts of a primary net cannot go astray.

To prove the theorem, we explore the space of reducts of $[P]$ and its confluence properties. According to §5.3, the possible redexes of $[P]$ are of the form $\mathsf{c.s} \bowtie \mathsf{c.o}$, $\mathsf{c.s} \bowtie \mathsf{u.d_1}$ and $\mathsf{c.r} \bowtie \mathsf{p.i}$. According to §5.4.1, the former two cause $\mathsf{c.o/u.d_1}$ to be transferred (committed) from $\mathsf{c.s}$ to $\mathsf{c.c}$, at which point the latter redex becomes applicable. We will call c nodes with something on $\mathsf{c.c}$ *committed channels*.

A committed channel with a $\mathsf{p.i}$ on its $\mathsf{c.r}$ port may choose to short-circuit one of its commitments with $\mathsf{p.i}$, and become non-committed if it has no more commitments, or may choose to stay committed. Lemma 5.4.3.2[23] applies to the redex formed of the commitment and $\mathsf{p.i}$, so we may reduce it immediately to the RHS. This will unblock the subordinate processes of c and p (if any) as well, and leave us with an "quasi primary" net, one that may eventually contain committed channels. We can

Thus, it turns out that the structure of the reduct space is quite simple:

- It may contain channels in various stages of commitment.

[22] With uncommitment.

[23] And an analogous lemma without u, and a direct $\mathsf{c} \bowtie \mathsf{p}$ cut instead.

- It may contain parts that are in the process of link migration and unblocking, but by confluence we can complete the process in essentially only one way.

Note The following

Conjecture 5.5.1.4 *For every* $[P] \stackrel{\tau}{\Rightarrow} [P']$ *holds* $P \Rightarrow P'$.

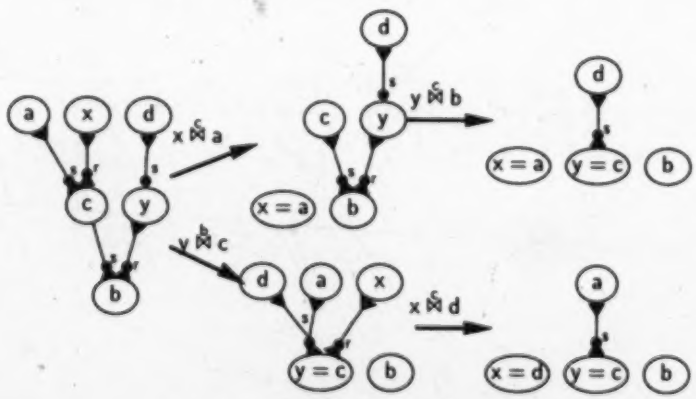
while true, is not sufficient to guarantee well-behavedness of our translation, because it leaves the possibility of "garbage" reductions starting from $[P]$ that never lead to a primary net. We could prove this statement by tracing where commitments are consumed, completing at these checkpoints, and relating the result to single-step process reductions.

5.6 Discussion: Why Multiple Principal Ports?

Throughout this thesis, we have strived to keep our translations as simple and intuitive as possible. This desire for simplicity sometimes conflicts with a desire for "minimality" in a certain sense. For example, it would have been more satisfying to implement the π-calculus in INMP or INMPP alone, not in MIN=INMP+INMPP, our inter-representation results between INMP and INMPP notwithstanding. It is quite clear that because a name can be related in the same role to several other names (*e.g.* output object of several output prefixes), we need INMP. We tried to find a translation in INMP alone, but failed.

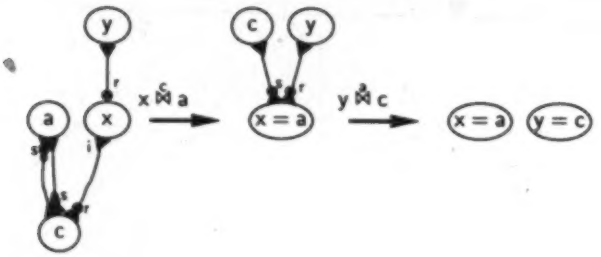
We now explain why we needed INMPP, namely why we needed both ports c.s and c.o to be principal, or why there are times when a channel needs to be attentive on both ports.

On one hand, o needs to be attentive all the time when there is at least one link to it, because link immigration through that port may bring in new s and r links, thus enriching the communication capabilities of c. Even if c is ready to perform a send/receive (there are links at both s and r), it would be incorrect to force it to do so by making o non-principal, because then c would miss some of the reductions it should be able to perform. For example, in $c!a, c?x, b!c, b?y, y!d$, c can either perform the communication x $\overset{c}{\bowtie}$ a, or it can first immigrate the links of y through channel b, and then do the communication x $\overset{c}{\bowtie}$ d. [⋈ 5.22]



⋈ 5.22: Why Port o Needs to be Principal.

On the other hand, we cannot postpone activating the s port until there are no links on the o port, because an interaction on s may enable one at o. Consider the translation of $c!a, a!c, c?x, x?y$. [⋈ 5.23]



⋈ 5.23: Why Port s Needs to be Principal.

By a construction similar to our inter-representation result of §4.6, we could have limited MIN_π to pure INMP, but we use the extended system MIN=INMP+INMPP for convenience.

5.7 Related Work

The only work which deals with an application of IN to a process calculus (that we are aware of) is the unpublished paper (Gay, 1991). It translates a confluent fragment of CCS into INs. The restriction to confluence is not arbitrary, it is due to the essential confluence of conventional IN.

5.7.1 π-nets and Interaction Diagrams

Since the very inception of the π-calculus, Flow Graphs were sometimes used to represent it graphically (Milner *et al.*, 1992). More recently, two other graphical formalisms were introduced, *π-nets* (Milner, 1994, 1993) and *Interaction Diagrams* (Parrow, 1995). Our translation is similar to these constructions, but we also implement synchronization in a distributed manner, using only local interaction and not the global synchronization mechanism called *boxes* which is used in those other formalisms. Our construction is more similar to INs because nodes have separate ports and we use only dyadic interaction.

These formalisms give additional insights into the finer computational structure of the π-calculus, but still leave some of the computational structure implicit and use names essentially. Namely, they represent prefix through the use of boxes (non-local computation elements). Nodes bear the names/numbers of the π-calculus names they represent, and communication manipulates these names/numbers "at runtime".

(Parrow, 1995, sec. 10) suggests that synchronization may be implemented locally by increasing the arities of every channel, but unfortunately the sketch given appears to be incorrect (or correct only in a limited setting). Parrow proposes to implement $a?\vec{x}.P$[24] by adding a control channel b to every output prefix $d!\vec{y}$ appearing in P, and making $a?\vec{x}$ instantiate these channels when it reduces. Input prefixes $d?\vec{y}$ also get an auxiliary channel, but it is not "hooked up" to $a?\vec{x}$, its purpose being only to match the control channel of $d!\vec{y}$. This indeed stops internal communications of $d?\vec{y}$, but it does not stop communications with the outside of P. Therefore the construction only works if all subjects in P are private channels. Nor will it be correct to block *all* interactions on d, because in $a?x.(d!y, d?y), d!z, d?z$, the prefixes $d!?y$ must wait, but $d!?z$ may proceed.

5.7.2 Concurrent Combinators

More recent is the work on Concurrent Combinators (cc) (Honda and Yoshida, 1994a,b), which "analyzes away" the prefix constructions of the π-calculus (input, output and replication prefix) and represents all of their expressive power in a finite system of atomic combinators. This work does the same for the foundations of concurrent computation as the work of Curry *et al.* does for the

[24] Here $\vec{x}$ denotes a tuple of names.

foundations of sequential functional computation. A graph representation of the same system is developed in (Yoshida, 1994) and is named *Process Graphs*. We presented the cc system in §3.4.1.

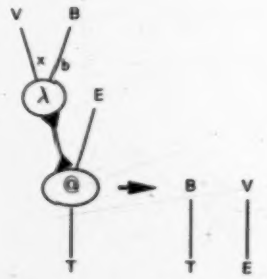
The work on cc was a big inspiration to us for the work presented in this chapter. cc can be seen as a dual graph representation of our construction. For example, where MIN_π represents $a!b$ as two nodes a, b and a connecting edge, cc represents the same process as one *message* node $M(a,b)$ with two links a, b. However, we perceived the following shortcomings of cc, and tried to address them in this chapter:

- cc are more complicated since they use "aggregate" nodes, *i.e.* nodes corresponding to configurations of π-calculus channels. For example, cc uses a node $S(u,v,w)$ corresponding to $u?x.v?y.w!y$. The genesis of the combinators and the corresponding rules is not obvious, and the proof that the set of combinators is closed with respect to process constructors is highly non-trivial. In contrast, our (main) node types correspond directly to π-calculus channels and some blocking machinery, and our rules come naturally.

- cc are inspired by combinatory term rewriting rather than graph rewriting. As a consequence, short-circuit edges are not allowed in the RHS of rules. For example, the typical IN rule corresponding to β-reduction in the λ-calculus

$$@(\lambda v.b, e) \to b[a/v]$$

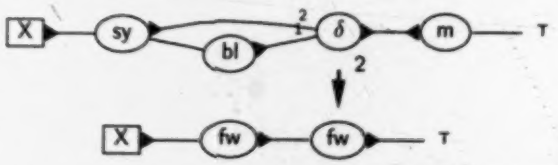
is not allowed in cc. [⋈ 5.24]



⋈ 5.24: β-Reduction as an Interaction Rule.

(Here the two nodes are *lambda abstraction* λ and *application* @, and the free edge names stand for Variable, lambda Body, argument Expression, and root Term.)

- Therefore one is forced to use *forwarders* whose role is purely bureaucratic: to disappear when a message arrives at their primary port, connecting it to their auxiliary port. Notwithstanding their trivial nature, effort has to be expended for their bookkeeping. Furthermore, one doesn't always have a term reduce to the "expected" term. For example, when the translation of $c?x.X$ (which consists of a duplicator δ, left binder bl, synchronizer sy and the translation of X) interacts with a message m, it reduces in 2 steps not to X, but to X "preceded" by two forwarders fw. [⋈ 5.25]



⋈ 5.25: Interaction of Prefix $c?x.X$ with a Message in cc.

Thus one has to be content with a reduction "up to" a certain simulation relation $\succ$ that effaces forwarders (see *e.g.* (Honda and Yoshida, 1994a, Theorem 4.4)).

- The translation of π-calculus into *cc* is asymmetric, using only one constructor (the *message*), and several destructors that come from analysis of the input prefix. The use of forwarders only makes this asymmetry stronger. We believe that a large part of the complexity of the translation can be attributed to this asymmetry, and that a more symmetric translation results in a simpler, or at least more natural translation.

- The translation from π-calculus to *cc* is not *uniform* (Palamidessi, 1997), in that $[\![P,Q]\!] \neq [\![P]\!],[\![Q]\!]$ (it uses a *duplicator* δ to multiplex an unblocking signal to the two components), and $[\![0]\!] \neq \emptyset$. Our translation is uniform.

- The translation can be exponential (our translation is linear). For example, the translation of

$$c?x_1.x_1?x_2.\ldots.x_{n-1}?x_n.x_n!v$$

is a process graph with $(3^{n+1}-1)/2$ nodes. The reason for such an explosion is that (almost) every node $X \in G$ in the translation of $x_i?x_{i+1}.G$ is surrounded by 3 new nodes (see $\bowtie$ 5.25) — a duplicator δ, a binder bl and a synchronizer sy.

This intractability can probably be eliminated through a more refined translation, one that does not deliver an activation signal from a reception to every node of the receiving graph, but only to some *controlling* nodes (like our $\mathcal{B}[\![P]\!]$). But we trace at least part of the reason to the use of forwarders.

On the other hand, *cc* have the advantage over our construction of being a combinatory system for the π-calculus, in the sense that they can be expressed in π and they form a closed system. Our MIN_π is a system external to π, and furthermore one that is not yet studied in depth.

We have shown in §4.7.4 that *cc* can be represented in INMP. Therefore, it should not be a great surprise that the π-calculus is representable in INMP. However, we presented here a completely new and direct translation from the π-calculus to INMP that has several nice properties: linear complexity, distributed synchronization, explicit representation of all computation elements.

5.8 Future Work: Replication and Choice

Two important π-calculus constructors are missing from our consideration, namely *replication/ recursion* and *choice (sum)*. These are the constructions that give the π-calculus full computing power (loops and conditionals). We hope to address these aspects of the π-calculus in future work.

Replication (the duplication of a process) and choice (the discarding of alternative processes) are similar to Linear Logic (LL) contraction and weakening respectively, which in the traditional Proof Net theory of LL are implemented as non-local operations through the duplication and erasure of "boxes". It would not be hard to postulate replication and choice in a similar way in this work, however we would like to preserve the local character of MIN at any price. Recent developments in the area of optimal lambda reduction (Gonthier *et al.*, 1992b) and sharing graphs (Guerrini *et al.*, 1997) demonstrate how one can present Proof Nets in a completely local fashion. The interaction of these "structural boxes" and the blocking "layers" that we use is quite subtle.

Chapter 6

Conclusion

The work reported in this thesis is quite exploratory in nature: we extend a well-known and elegant system (IN) in various ways, and explore the expressive power of the resulting systems. Before our extensions can become established, it is also necessary to study the nature (internal theory) of the resulting systems. Such study is outside the scope of this thesis, and is a good candidate for future research.

6.1 Directions for Future Work

The following questions are natural. Apart from posing them, here we allow ourselves some commentaries, and even make some pure conjectures about the possible answers.

How far have we strayed from INs? The answer is different for the different extended systems.

INMR don't stray far enough. As we show in §4.2, they can't capture non-determinism in a reasonable way, and the reason is that they can't represent the notion of an agent being connected and responsive to several other agents.

INMPP allow a node *more* interaction capabilities than it may have in IN, but since one interaction may preclude another, these interaction capabilities are *less* permanent. This complicates the interaction patterns of INMPP. On the other hand, they preserve the linear structure of INs (ports are connected only in pairs), so some structural results may be re-gained. Bawden (1986, 1992) studies a formalism not far from INMCC, and similarly to us uses a more pragmatic approach.

INMP give up the linearity of IN, but since we insist that edge transfers be done explicitly and one by one, we still capture all steps of a computation. This keeps INMP simple for implementation; the main difference between INMP and IN in this aspect is that an INMP port needs to keep a list of pointers to neighbor nodes, as opposed to one pointer. The connectivity structure of an INMP net is more complex than that of INMPP, but the activation patterns of INMP are simpler. INMP and INMPP extend IN in orthogonal directions, and we would expect that they stray from IN "to the same extent". INMP seem to be the most natural and useful approach for the modeling of multi-connected concurrent agents.

INMC in our opinion stray too far from IN. Essentially they allow *multi-casting* between several nodes. We prefer to keep the multi-connectedness aspect explicit, by associating it with a "reified" computational entity such as a multiport, as opposed to an abstract entity such as a connection point. In an implementation of INMC, you would have to introduce an entity corresponding to a connection point, and implement the merging of two connection points, like we did in §4.7. But we think that is better to represent these

elements explicitly in the formalism, so that the computational results that we obtain have a better relation to reality.

Which of the theoretical results for IN may hold in our systems? We have consciously given up confluence, but we find it very important to still identify confluent fragments of a non-deterministic IN system and confluent fragments of reduction sequences. Such fragments allow us to analyze more easily the deterministic fragments of computation, and then focus our analysis on the points where a real choice is made. Such analyses give us some understanding of an important problem in computing: the integration and interplay of functional and object-oriented (stateful, concurrent) computation.

Can one identify useful deadlock-free fragments? The most difficult aspect of this question is to formulate what exactly is a deadlock in an extended IN system. The "rules of the game" change in extended INs, since it is not anymore the case that a node has a single communication capability. We think that it may not be appropriate to ask this question in general, and that rather one should focus on application-specific aspects of it. For example, if a particular concurrent object-oriented system is translated to extended INs, can we guarantee that a request will always be answered, that an object will never be blocked, etc.

Deadlock-freedom is only one criterion of "goodness" for a concurrent system. Others are absence of divergent computations, atomicity of various computation fragments, etc. Various such properties are important when studying implementation translations (encodings) between systems, and we have studied some of them in §5.2.

Is there a universal combinatory system for non-deterministic INs? We have spent some time on modeling concurrent object-oriented programming with extended INs, studying the various translations of COOP languages to the π-calculus (Jones, 1993b,a; Pierce and Turner, 1994; Ravara and Vasconcelos, 1996), and studying linear logic programming. From these experiences we gained an understanding of doing "programming" with extended INs, representing the code for such programs explicitly, replicating that code, transferring it between agents, and manipulating it in other ways.

The combinatory translations of Lafont (1995b) and Gay (1994) amount to pretty much the same: the rules of a source system S are encoded as node configurations in a universal system IC (code), and this code is then replicated for the translation of every source node. Our experience tells us that indeed a universal non-deterministic IN system should exist.

Another question to explore here is this: Lafont's system IC is not "modular", in the sense that adding a new rule or a new node type to S changes (complicates) the translations of all nodes, even ones to which the rule does not apply, and ones that will never come in contact with the new node. The reason is that the code for the entire system S is replicated at every node. It would be interesting to explore optimizations of combinatory translations for systems that can be partitioned in some way. Specifically, an encapsulated object-oriented system must allow translation to a system where an "object" node type carries only the code of the methods of that object, but not others. If an object node needs to create a new object, it could consult an "object factory" node for that object class, etc.

In addition, it is natural to compare non-deterministic INs to other related formalisms. We don't have much to say here, except to give a few pointers.

Graph rewriting IN are a special case of graph rewriting, so it can be expected that as we extend IN, we may step into the territory of useful results coming from general graph rewriting. Much of of the theory of graph rewriting focuses on abstract algebraic characterizations of the systems, often relating to category theory. We think that such approaches are useful in identifying classes of systems and finding new extensions (for example Banach (1995) identifies various extensions of Lafont's "simple nets" using a similar approach), but stray far from more

pragmatic applications such as modeling various computing formalisms. An example of such a work is (Engels and Schürr, 1995) which studies ways of working with "subroutines" in graph grammars.

Term rewriting and Combinatory rewriting systems Although unlike IN, these systems are one-dimensional, it is possible to compare them, since by using multiple occurrences of the same variable, one can represent "pointers" (the analog of "connections") in term rewriting and combinatory rewriting. Fernández and Mackie (1996b) lay out some foundational work on relating conventional IN to restricted classes of term rewriting systems. The same authors introduce a non-deterministic generalization of IN in (Fernández and Mackie, 1996a) and compare them to more general term rewriting. That generalization is comparable to INMP, but also introduces some new entities (*memory* and pseudo-edges), and we find that it strays too far from IN.

We have only scratched the surface of the relation between non-determinism and Interaction Nets. We hope to have shown clearly the expressive power of the extended systems, and that we have raised some interest in non-deterministic INs, so others may study the properties of these systems.

6.2 Contrubutions

The most important contribution of this thesis is the demonstration that some simple and intuitive additions to IN give them considerable expressive power and allow them to represent complex non-deterministic systems such as the π-calculus. We can formulate this succinctly as the slogan

$$\text{Concurrency} = \text{interaction} + \text{non-determinism}.$$

It came to us as a pleasant surprise that adding simple non-deterministic extensions to IN was sufficient to capture the π-calculus ***in a straightforward way*** (compare our construction of §5.2 to the complicated constructions of Honda and Yoshida (1994a,b)), which furthermore exhibits nice properties (linear complexity, uniformity of translation, linear number of intermediate steps). Furthermore, our translation allowed us to exhibit some shortcomings of the traditional formulations of the π-calculus, namely global synchronization and the overloading of prefix with many roles.

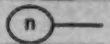
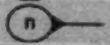
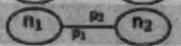
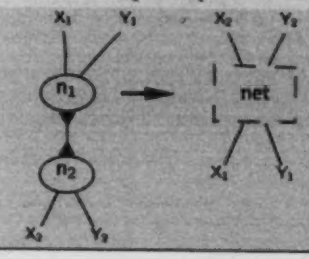
Another contribution of the thesis is the formulation of several different IN extensions, exploration of various examples using them, and the study of their relative expressive power. Another surprise was that it didn't matter much precisely which extension was used (with the exception of INMR), a kind of "personal Church-Turing thesis" regarding concurrent computation.

We sign off here, gentle reader, with the hope that this work has sparked sufficient interest for someone to continue the study of non-deterministic Interaction Nets, and to gain further insights into the nature of concurrency.

Appendix A

Notations

We use the following textual and graphical notations to represent concepts of IN (both conventional and the non-deterministic variations that we consider).

Notation	Description
$n.p$	Node and named port
$n_1.p_1 - n_2.p_2$	Edge between two named ports
$n_1.p - n_2$	Edge between a named port and the principal port of n_2
$n_1 \bowtie n_2$	Edge between two principal ports (cut)
IN	**Interaction Nets in general**
	Node (agent)
	"Abstract" node
	Auxiliary port (and edge)
	Principal port
	Cut (redex)
	Named ports
	Net
	Interaction rule

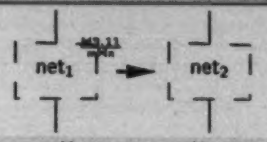
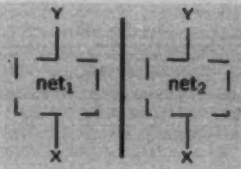
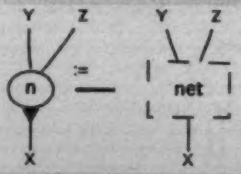
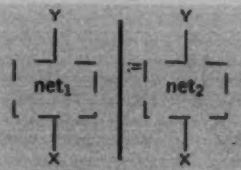
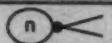
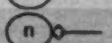
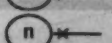
Notation	Description
	Reduction due to the cut m ⋈ n and numbered rule ⋈ 2.11
	Separator between two parts of a diagram (a merely visual element)
	Node definition
	net_1 is translated to net_2
INMP	**IN with MultiPorts**
	(Auxiliary) Multiport
	Principal multiport
	Multiport with no connections
	Multiport with exactly one connection
	Multiport with at least one connection
INMC	**IN with Multiple Connections**
	Connector

Table A.1: Notations

Appendix B

Technical Note: How Was This Thesis Produced

Like most academic writings of late, this thesis was produced with the TEX typesetting system and the LATEX 2_ε macro package. For writing, I used the One True Editor `xemacs`, with the *AuCTeX* and `bib-cite` add-on packages. I formatted tables using the `array`, `longtable` and `colortbl` styles. I used the `diagrams.tex` macro package by Paul Taylor for a few diagrams.

The IN figures in this thesis were produced using the following components:

- A heavily-customized version of the graph layout program `dot`, which is part of the GraphViz 1.21 package from AT&T. It is available from the URL `www.research.att.com/orgs/ssr/book/reuse`.
- A compiler called `in` from a textual notation for the description of INs to the graph description language of `dot`. I wrote this compiler in Perl, it consists of 11 packages (object classes) and 1500 lines of tightly-packed code.
  - I used `flex` to generate a scanner (lexer) and linked it to Perl using the XS interface.
  - I used a modified version of `byacc` (Berkeley `yacc`) that can produce Perl code to generate a parser.
- A TEX style file called `in.sty` that writes out IN descriptions embedded in TEX source, and replaces them with the corresponding figures. I also used the `psfrag` style to replace textual labels in the figures with text generated by LATEX (including Greek, subscripts and superscripts).

For example, to produce the figure ⋈ 2.3, I typed the following textual description right in the middle of my TEX source:

```
\begin{IN}{in-append}{The List ``Append'' Operation.}
  GLOBAL app:r(x!,y);
  T=app(nil,Y) => T=Y ;;;;[L=3]
  T=app(cons(E,X),Y) => T=cons(E,app(X,Y))
\end{IN}
```

The `in` format is quite simple and intuitive, but unfortunately one often has to give various "hints" to `dot`, or else it cannot produce high-quality layout.

On a PC with Pentium II 199MHz CPU and 64Mb RAM running Linux 2.0.34, it takes about 100 seconds (8 seconds of CPU time) to process the 175 INs in this thesis with `in` and `dot`. I found that in order to process this thesis with TEX and view it with Ghostscript, I had to increase the

pool size of TEX from 124000 to 150000 words, and to increase the operand stack of Ghostscript from 800 to 2400 words, and then to recompile both programs.

If you would like to use any of these components for your own writing, please contact me by email at `vladimir@workscape.com`.

Bibliography

R. Banach. The algebraic theory of interaction nets. Technical Report UMCS-95-7-2, University of Manchester, 1995.

A. Bawden. Connection graphs. In *Symp. on Lisp and Functional Programming*, pages 258–265. 1986.

A. Bawden. *Linear Graph Reduction: Confronting the Cost of Naming.* Ph.D. thesis, MIT LCS, 1992.

V. Danos and L. Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28:181–203, 1989.

G. Engels and A. Schürr. Encapsulated hierarchical graphs, graph types, and meta types. In A. Corradini and U. Montanari, editors, *Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRAGRA'95)*. Elsevier, Volterra (Pisa), Italy, 1995. *Electronic Notes in TCS* 2.

M. Fernández. Type assignment and termination of interaction nets. *Mathematical Structures in Computer Science (MSCS)*, 1998. To appear in Special Issue dedicated to the Intl. Conf. on Logic and Models of Computation, 1996.

M. Fernández and I. Mackie. From term rewriting to generalised interaction nets. In *Programming Languages, Implementations, Logics, and Programs (PLILP'96)*, number 1140 in LNCS, pages 319–333. Aachen, Germany, 1996a.

M. Fernández and I. Mackie. Interaction nets and term rewriting systems. In *Trees in Algebra and Programming (CAAP'96)*, number 1059 in LNCS, pages 149–164. Linköping, Sweden, 1996b.

M. Fernández and I. Mackie. Coinductive techniques for operational equivalence of interaction nets. In *Symp. on Logic in Computer Science (LICS'98)*. 1998.

S. J. Gay. Translating confluent CCS into interaction nets. 1991.

S. J. Gay. Combinators for interaction nets. In C. Hankin, I. Mackie, and R. Nagarajan, editors, *Workshop on Theory and Formal Methods of Computing*. Department of Computing, Imperial College, 1994.

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Principles of Programming Languages (POPL'92)*, pages 15–26. 1992a.

G. Gonthier, M. Abadi, and J.-J. Lévy. Linear logic without boxes. In *Logic in Computer Science (LICS'92)*, pages 223–234. IEEE Computer Society Press, Santa Cruz, CA, 1992b.

S. Guerrini. A general theory of sharing graphs. Technical Report IRCS-97-04, University of Pennsylvania, 1997.

S. Guerrini, S. Martini, and A. Masini. Coherence for sharing proof-nets. Technical Report IRCS-97-03, University of Pennsylvania, 1997.

C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, 1st edition, 1985. ISBN 0-13-153271-5.

K. Honda and N. Yoshida. Combinatory representation of mobile processes. In *Principles of Programming Languages (POPL'94)*, pages 348–360. Portland, Oregon, 1994a. ISBN 0-89791-636-0.

K. Honda and N. Yoshida. Replication in concurrent combinators. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computing Science (TACS'94)*, number 789 in LNCS, pages 786–805. Sendai, Japan, 1994b.

C. B. Jones. A pi-calculus semantics for an object-based design notation. In E. Best, editor, *4th International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *LNCS*, pages 158–172. Hildesheim, Germany, 1993a.

C. B. Jones. Process-algebraic foundations for an object-based design notation. Technical Report UMCS-93-10-1, University of Manchester, 1993b.

Y. Lafont. Interaction nets. In *Principles of Programming Languages (POPL'90)*, pages 95–108. ACM, San Francisco, CA, 1990.

Y. Lafont. The paradigm of interaction (short version). 1991.

Y. Lafont. From proof nets to interaction nets. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic, Proc. of the Workshop on Linear Logic, Cornell University, June 1993*, number 222 in London Mathematical Society Lecture Notes Series, pages 225–247. Cambridge University Press, 1995a.

Y. Lafont. Interaction combinators. Preprint 95-29, University of Marseilles, 1995b.

J. Lamping. An algorithm for optimal lambda calculus reductions. In *Principles of Programming Languages (POPL'90)*, pages 16–30. San Francisco, CA, 1990. Also Xerox PARC technical report, 1989.

I. Mackie. A lambda-evaluator based on interaction nets. In C. Hankin, I. Mackie, and R. Nagarajan, editors, *Workshop on Theory and Formal Methods of Computing*. Department of Computing, Imperial College, 1994.

R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92. Springer-Verlag, Berlin/New York, 1980.

R. Milner. An action structure for the synchronous π-calculus. In Z. Esik, editor, *Fundamentals of Computation Theory (FCT'93)*, number 710 in LNCS, pages 87–105. Szeged, Hungary, 1993.

R. Milner. Pi-nets: a graphical form of π-calculus. In *European Symposium on Programming (ESOP'94)*, number 788 in LNCS, pages 26–42. 1994.

R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.

U. Nestmann and B. C. Pierce. Decoding choice encodings (extended abstract). In *Intl. Conf. on Concurrency Theory (CONCUR'96)*, number 1119 in LNCS, pages 179–194. 1996. Full version is ERCIM Research Report 10/97-R051, August 1997, 51pp.

C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *Principles of Programming Languages (POPL'97)*, pages 256–265. 1997.

J. Parrow. Interaction diagrams. *Nordic Journal of Computing*, (2):407–443, 1995. Earlier version appeared in *A Decade of Concurrency: Reflections and Perspectives. REX School and Symposium*, J.W. de Bakker, W.-P. de Roever and G. Rozenberg (ed), June 1993, LNCS 803; and as SICS Research report R93:06.

B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In T. Ito and A. Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP'94)*, number 907 in LNCS, pages 187–215. Sendai, Japan, 1994.

A. Ravara and V. Vasconcelos. Operational semantics and type system for GNOME on Typed Calculus of Objects. In *ECOOP'96 Workshop on Proof Theory of Object-oriented Programming*. 1996.

N. Yoshida. Graph notation for concurrent combinators. In T. Ito and A. Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP'94)*, number 907 in LNCS, pages 393–412. Sendai, Japan, 1994.